



**FERRAMENTA PARA CONSTRUÇÃO DE LINHA
DE PRODUTOS NO ECLIPSE**

Aluno: **Alexandre Torres Vasconcelos** (atv@cin.ufpe.br)
Orientador: **Paulo Henrique Monteiro Borba** (phmb@cin.ufpe.br)

Recife, agosto de 2005.

Resumo

A atividade de portar jogos para celulares de um aparelho para outro com configurações bastante diferentes tem aumentado na indústria de software, pois a diversidade de dispositivos lançados no mercado pelos fabricantes vem crescendo constantemente. Visando atingir o maior número de aparelhos possíveis com um mesmo jogo, as empresas de desenvolvimento de jogos têm estudado técnicas para separar de forma mais eficiente as diferenças de código existentes entre cada versão de uma plataforma, facilitando assim o porte e a manutenção desses aplicativos. Técnicas de modularização de software aliadas ao paradigma de Orientação a Objetos têm se mostrado ineficientes para separar essas diferenças, que se encontram geralmente espalhadas e entrelaçadas com o núcleo do jogo. Na prática, o software acaba tendo que ser re-implementado e posteriores modificações nos requisitos têm que ser aplicadas a todas as versões desenvolvidas. O objetivo do nosso trabalho é apresentar uma ferramenta que facilita o processo de porte de jogos, extraindo as variações de código das diversas plataformas utilizando Programação Orientada a Aspectos (POA). Aliada a uma abordagem extrativa e incremental, juntamente com técnicas de Linha de Produtos, a POA apresenta-se como uma solução viável para resolver esse atual problema da indústria de jogos.

Palavras-chave: portabilidade de jogos, plataformas de celulares, linha de produtos, orientação a aspectos.

Agradecimentos

Ao meu orientador, Paulo Borba, pela oportunidade de contar com seu apoio, acompanhamento e confiança para a construção deste trabalho.

A Vander Alves, pelo acompanhamento, paciência e suporte dado nas etapas mais difíceis do projeto.

A Pedro Osandy, Ivan Cardim e Heitor do Carmo, que colaboraram com suas experiências na definição dos requisitos do nosso projeto.

A Matt Chapman, Adrian Colyer e Andrew Clement, desenvolvedores do AJDT, pela ajuda na compreensão dos recursos mais obscuros dessa ferramenta.

Aos desenvolvedores que fazem parte da Comunidade Eclipse, pela construção de uma plataforma extremamente robusta para a execução da nossa ferramenta.

Aos professores do Centro de Informática, pela formação e pelo apoio que me deram sempre que solicitados.

A minha família e amigos, como os já citados acima, pelo apoio e confiança dados durante diversas etapas da minha vida e formação acadêmica.

*O ferro enferruja com a falta de uso;
a água estagnada perde sua pureza...
Da mesma forma, a falta de ação mina o vigor da mente*
— LEONARDO DA VINCI

Índice

1. INTRODUÇÃO	7
1.1. ORGANIZAÇÃO DO TRABALHO	8
2. CONCEITOS IMPORTANTES	9
2.1. ECLIPSE	9
2.1.2 Arquitetura da plataforma	10
2.2 JDT E BIBLIOTECAS DE MANIPULAÇÃO DE ASTs	12
2.2.1 org.eclipse.jdt.core.dom	13
2.2.2 org.eclipse.jdt.core	15
2.3 PROGRAMAÇÃO ORIENTADA A ASPECTOS (POA)	15
2.3.1. AspectJ	17
2.3.2. AJDT	18
3. PROJETO DA FERRAMENTA	19
3.1. PROBLEMÁTICA	19
3.2. A FERRAMENTA	21
3.3. <i>REFACTORINGS</i>	23
3.3.1 Refactoring Extract Method to Aspect	23
3.3.2 Refactoring Extract Resource to Aspect - after	24
3.3.3 Refactoring Extract Context	26
3.3.4 Refactoring Extract Before Block	27
3.3.5 Refactoring Extract After Block	28
3.3.6 Refactoring Change Class Hierarchy	29
4. ARQUITETURA DA FERRAMENTA	30
4.1. REFACTORING EXTRACT METHOD TO ASPECT	30

4.1.1 Padrão visitor do JDT	32
4.1.2 Arquitetura do plug-in	34
4.1.3 Processo de Refactoring	37
5. IMPLEMENTAÇÃO	41
5.1 IMPLEMENTAÇÃO DO <i>REFACTORING EXTRACT METHOD TO ASPECT</i>	41
5.2 IMPLEMENTAÇÃO DOS OUTROS <i>REFACTORINGS</i>	47
5.2.1 Refactoring Extract Resource to Aspect - after	47
5.2.2 Refactoring Extract Context	49
5.2.3 Refactoring Extract Before Block	50
5.2.4 Refactoring Extract After Block	51
5.2.5 Refactoring Change Class Hierarchy	51
6. CENÁRIOS DE USO	53
6.1. <i>REFACTORING EXTRACT METHOD TO ASPECT</i>	53
6.2. <i>REFACTORING EXTRACT RESOURCE TO ASPECT - AFTER</i>	53
6.3. <i>REFACTORING EXTRACT CONTEXT</i>	55
6.4. <i>REFACTORING EXTRACT BEFORE BLOCK</i>	56
6.5. <i>REFACTORING EXTRACT AFTER BLOCK</i>	58
6.6. <i>REFACTORING CHANGE CLASS HIERARCHY</i>	59
7. CONCLUSÕES E TRABALHOS FUTUROS	61
REFERÊNCIAS BIBLIOGRÁFICAS	62

1. Introdução

Empresas de desenvolvimento de jogos para dispositivos móveis, no mercado atual, têm a necessidade de disponibilizar seus produtos para uma maior variedade de aparelhos possíveis, de forma a maximizar seus lucros. Em contrapartida, para cada família de aparelhos, é necessária uma implementação diferente de um mesmo jogo. Essa problemática decorre da grande diversificação desses dispositivos no mercado, cada um com suas respectivas limitações **[Cardim et al., 2005]**, como:

- ☑ Diferentes tamanhos de tela, número de cores, tamanho dos *pixels*, sons e *layouts* de teclado;
- ☑ Diferentes disponibilidades de memória de execução e armazenamento;
- ☑ Existência de API's proprietárias e pacotes opcionais;
- ☑ *Bugs* específicos para cada dispositivo;
- ☑ Internacionalização (necessidade de traduzir o jogo para diversas línguas).

Para oferecer diversas versões dos seus jogos, as empresas utilizam o processo de Porte (*porting*) como solução, adaptando a implementação original para os demais dispositivos. Esse processo atualmente é executado de maneira pouco eficiente, pois quase sempre separar código de certas características dos aparelhos não é uma tarefa trivial, já que está espalhado pela aplicação e freqüentemente entrelaçado com outras características **[Alves, 2005]**. Não obstante, os fabricantes lançam aparelhos diferentes, visando públicos distintos, em um intervalo cada vez menor de tempo. Esse fato torna o processo de portar jogos uma tarefa crítica, pois, em um curto espaço de tempo é preciso identificar e isolar as variações das diversas versões dos produtos **[Cardim et al., 2005]**. Isso tem levado, por exemplo, ao surgimento de empresas especializadas em *porting* **[Tira Wireless, 2004]**.

A literatura dispõe de algumas soluções para esse problema, mas poucas delas foram validadas na indústria **[Cardim et al., 2005]**. Mais recentemente, no entanto, pesquisadores têm realizado estudos de caso e experimentos visando a melhoria do processo de *porting*. Dentre as soluções estudadas, Linha de Produtos de Software **[Clements, 2002]** é uma opção para reduzir os custos de desenvolvimento e agilizar o processo de porte, levando os produtos para o mercado em um tempo satisfatório **[Alves, 2004]**.

Em particular, foi proposta uma abordagem extrativa e incremental para portar produtos existentes para outras plataformas, usando Linha de Produtos e Programação Orientada a Aspectos (POA) **[Kiczales, 1997]**. Tal pesquisa conseguiu identificar, através de um estudo de caso da indústria, alguns padrões de variação. A validação desses padrões de variação é realizada, em outro estudo de caso mais complexo, no trabalho de graduação de Heitor Vital do Carmo **[Carmo, 2005]**.

Nesse contexto, o presente trabalho apresenta o desenvolvimento de uma ferramenta que facilita o processo de porte dessas aplicações, utilizando conceitos de Linha de Produtos e Orientação a Aspectos. Ela é a primeira versão de um ambiente que será utilizado para portar jogos de maneira ágil e eficiente. Sua implementação consiste de diversas operações para a extração de variações no

código dos jogos para aspectos. Operações que foram definidas a partir dos padrões identificados nos estudos de caso.

1.1. Organização do trabalho

O capítulo 2 aborda conceitos que são importantes para o entendimento deste trabalho, como arquitetura da plataforma Eclipse, pacotes de manipulação de AST's do JDT, Orientação a Aspectos, *AspectJ* e AJDT.

O capítulo 3 trata do projeto da ferramenta, explicando como ela será utilizada no processo de *porting*, e detalha os padrões de variação que foram identificados no estudo de caso da indústria.

O capítulo 4 trata da arquitetura da ferramenta, explicando os padrões que foram utilizados, o funcionamento dos *plug-ins* e como estão organizados os pacotes do projeto e os diagramas de classe.

O capítulo 5 aborda a implementação, comentando todo o processo de desenvolvimento utilizando, eventualmente, exemplos de código de um dos *refactorings* e descrevendo o fluxo de funcionamento dos demais.

O capítulo 6 exhibe os cenários de uso possíveis da ferramenta, servindo também como um manual para a utilização da mesma.

O capítulo 7 conclui nosso trabalho, apresentando também as próximas etapas de implementação da ferramenta.

2. Conceitos importantes

Antes de discorrermos sobre o projeto, alguns conceitos precisam ser entendidos para a compreensão do trabalho. A seguir, analisamos a plataforma Eclipse, onde nosso projeto é executado, e sua arquitetura, que favorece o desenvolvimento de ferramentas como a nossa. Além disso, explicamos como estão estruturados os principais pacotes utilizados no projeto, que fazem parte do JDT, ferramenta que acompanha o Eclipse SDK. Por fim, daremos uma introdução à Programação Orientada a Aspectos e *AspectJ*, juntamente com a motivação para utilizar esse novo paradigma em diversos domínios da área de desenvolvimento de *software*, além do nosso.

2.1. Eclipse

O Eclipse é uma plataforma de desenvolvimento aberta e gratuita, designada para a construção de ambientes de desenvolvimento integrados (IDEs) que podem ser usados para a criação de aplicações tão diversas como *web-sites*, programas em linguagens Java™, C ou C++, aplicações para sistemas embarcados, entre outras [Clayberg & Rubel, 2004]. Seu público alvo é composto de desenvolvedores de aplicações ou de ferramentas de desenvolvimento. Este capítulo detalha aspectos mais interessantes para o segundo grupo.

A plataforma Eclipse possui uma série de componentes genéricos, como navegadores para acesso a arquivos usados na área de trabalho, editores de texto, entre outros. Todos eles podem ser reaproveitados na confecção de novos ambientes. Quando iniciada, a plataforma executa um processo de procura, integração e execução de módulos chamados *plug-ins*. Através da confecção desses módulos, desenvolvedores podem adicionar novas funcionalidades à plataforma, aproveitando, se necessário, os componentes genéricos já mencionados. De fato, a IDE que é exibida quando o Eclipse é inicializado nada mais é do que um conjunto de *plug-ins* executando sobre a plataforma.

Como o Eclipse foi desenvolvido para executar sobre a Máquina Virtual Java™ (JVM), a característica de portabilidade para uma grande variedade de sistemas operacionais é um dos pontos fortes da plataforma. Os *plug-ins* também devem ser desenvolvidos em Java™, aproveitando a popularidade dessa linguagem, e a integração deles é feita através de um arquivo XML (Figura 2.1), onde estão definidas interfaces de conexão entre os mesmos.

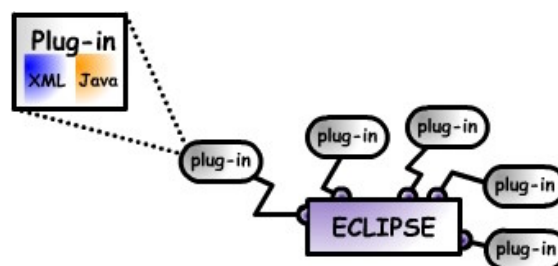


Figura 2.1: estrutura de um *plug-in*

O principal objetivo da plataforma Eclipse é oferecer aos desenvolvedores de ferramentas mecanismos e regras que permitam a implementação de funcionalidades para um ambiente em comum.

2.1.2 Arquitetura da plataforma

A arquitetura do Eclipse é exibida na Figura 2.2. Uma “Nova Ferramenta” nada mais é do que um conjunto de plug-ins.

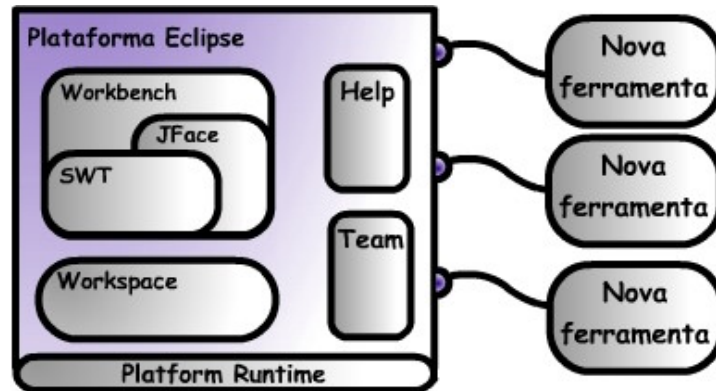


Figura 2.2: arquitetura da plataforma Eclipse

Platform Runtime

Geralmente, uma ferramenta simples é distribuída como um simples *plug-in*; uma ferramenta mais complexa, como vários *plug-ins*. Exceto pelo *kernel*, conhecido como *Platform Runtime*, todas as funcionalidades da plataforma estão disponíveis em *plug-ins*.

Um *plug-in* geralmente consiste de código Java™ dentro de uma biblioteca JAR, juntamente com arquivos de configuração e outros arquivos como imagens, templates, bibliotecas de código nativo, entre outros. Cada *plug-in* possui um arquivo *manifest* escrito em XML, declarando sua interconexão com outros *plug-ins*. Nele, “pontos de extensão” são declarados e suas “extensões” a outros “pontos de extensão” (no mínimo, uma “extensão”) de outros *plug-ins* são especificadas. Por exemplo, o *plug-in* do *workspace* declara um “ponto de extensão” para editores. Qualquer desenvolvedor pode conectar uma nova “extensão” a esse ponto e adicionar um editor para a linguagem C#, por exemplo, para a plataforma. Um “ponto de extensão” deve ter uma interface Java™ associada. Outros *plug-ins* que queiram utilizar esse ponto devem definir classes que implementam essa interface, além de declarar no *manifest* a “extensão” para esse ponto (Figura 2.3).

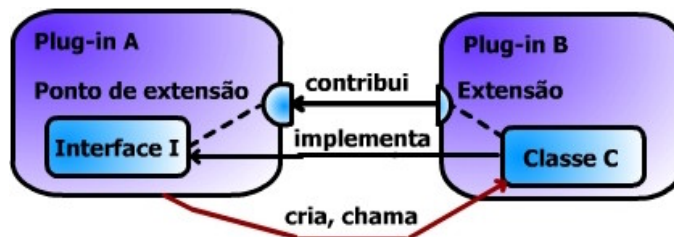


Figura 2.3: conexão de *plug-ins*

O processo de instalação de uma nova ferramenta é simples. Um usuário precisa apenas copiar os arquivos referentes aos plug-ins dessa ferramenta para uma pasta padrão do Eclipse. Ao inicializar, a plataforma descobre o conjunto de *plug-ins* disponíveis, lê seus arquivos *manifest* e cria um registro na memória desses arquivos. Em seguida, a plataforma casa as declarações de “extensão” com suas declarações de “ponto de extensão” correspondentes.

Para facilitar o desenvolvimento de *plug-ins*, o Eclipse SDK, conjunto de ferramentas que acompanham a plataforma Eclipse, oferece um Ambiente de Desenvolvimento de Plug-ins (PDE).

Workspace

O *workspace* consiste de um ou mais projetos, associados a diretórios do sistema de arquivos do usuário, sempre em sincronização com o mesmo. Cada projeto contém arquivos que são criados e manipulados pelo usuário, sendo acessíveis por qualquer ferramenta da plataforma. O *workspace* também oferece uma série de mecanismos, como histórico e marcadores que auxiliam na identificação de tarefas de implementação ou no processo de depuração do código.

Workbench e Ferramentas de Interface Gráfica

O *workbench* representa a interface gráfica do Eclipse. Ele oferece uma série de pontos de extensão para qualquer desenvolvedor adicionar novos recursos. Para isso, dois *toolkits* são utilizados:

- ☑ **SWT**: conjunto de bibliotecas gráficas baseadas em código nativo, mas com uma API independente de sistema operacional.
- ☑ **JFace**: conjunto de bibliotecas gráficas implementadas em SWT. Simplifica o desenvolvimento de componentes gráficos utilizados frequentemente.

Os componentes desses *toolkits* são totalmente compatíveis com o *workbench*, que também foi implementado utilizando essas ferramentas. Uma estrutura básica do *workbench* pode ser vista na Figura 2.4.

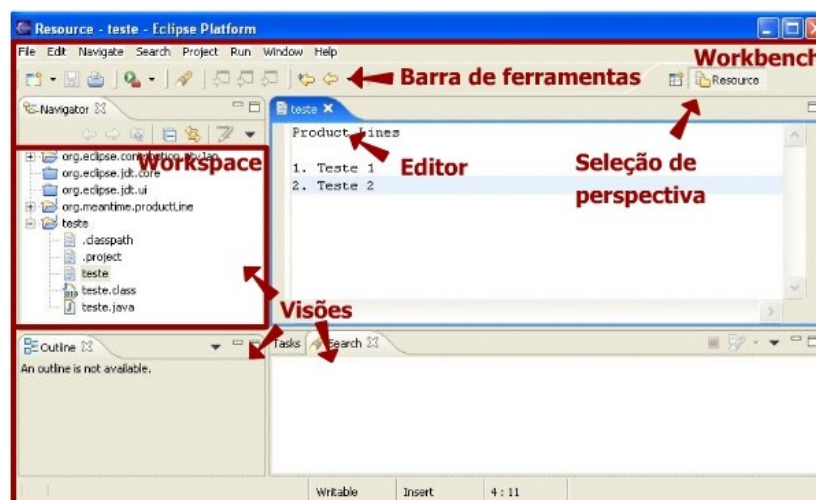


Figura 2.4: estrutura do workbench

Um *workbench* consiste de **visões** e **editores**. Através de **perspectivas**, podemos definir qual o posicionamento dos editores e quais visões aparecem na tela.

Utilizando **ações**, é possível adicionar dispositivos ao *workbench* que aguardam eventos do usuário para realizar alterações no estado de todos esses componentes (ex. executar um *refactoring* em uma seleção do editor, alterar o estado de uma estrutura de uma visão ou trocar de perspectiva). Uma nova ação pode ser adicionada em vários lugares do *workbench* através de botões no menu, na barra de ferramentas, em janelas *pop-up*, em janelas de visões, entre outros.

Editores permitem ao usuário abrir, editar e salvar documentos. Quando ativo, ele pode contribuir com um determinado conjunto de ações para a barra de menu e para a barra de ferramentas do *workbench*. A plataforma oferece um editor de texto padrão. Editores mais específicos podem ser implementados por outros plug-ins.

Visões oferecem informações sobre objetos que o usuário está manipulando no *workbench*. Por exemplo, a visão *Content Outline* exibe de forma estruturada o conteúdo de um editor que está ativo. Visões têm um ciclo de vida mais simples do que os editores. Modificações feitas em uma delas são salvas imediatamente, e as mudanças são refletidas no mesmo instante em outras partes relacionadas.

O *workbench* pode ter uma série de perspectivas. Apenas uma delas pode estar ativa de cada vez, determinando a configuração inicial de posicionamento do editor e quais as visões exibidas. Outras visões podem ser abertas e os editores podem ser re-posicionados depois que uma perspectiva é iniciada.

O *workbench* oferece pontos de extensão para a implementação de novos editores, visões, perspectivas e ações.

Os componentes *Team* e *Help* da Figura 2.2 não serão detalhados aqui por não serem importantes para o âmbito do projeto. O primeiro refere-se a ferramentas de auxílio ao trabalho em equipe (ex. repositórios de CVS), e o segundo trata de mecanismos para contribuição de documentação *online*.

2.2 JDT e Bibliotecas de Manipulação de ASTs

O Java™ *Development Tooling* (JDT) é uma ferramenta (conjunto de plug-ins) do Eclipse SDK que adiciona recursos à plataforma Eclipse para o desenvolvimento de aplicativos Java™. Além de possuir recursos básicos como visões que exibem código Java™ de forma estruturada, ações de busca, ações de alteração de código, perspectiva de testes com *JUnit*, entre outros, o JDT dispõe de uma série de pontos de extensão e bibliotecas que podem ser usados por desenvolvedores que queiram agregar mais funcionalidades à ferramenta.

Dentre o conjunto de bibliotecas, merecem destaque no trabalho apresentado a ***org.eclipse.jdt.core.dom***, que permite a navegação da árvore sintática (AST - *Abstract Syntax Tree*) de uma classe qualquer e a ***org.eclipse.jdt.core***, que auxilia na manipulação de projetos, pacotes e classes de forma estruturada.

2.2.1 org.eclipse.jdt.core.dom

Essa biblioteca possui uma série de classes que representam os nós de uma AST de um arquivo .java. Todos os nós são subclasses do tipo *ASTNode*. A raiz da árvore é o nó do tipo *CompilationUnit*, que representa o arquivo fonte .java. A estrutura de uma AST Java pode ser vista na Figura 2.5 e na Figura 2.6.

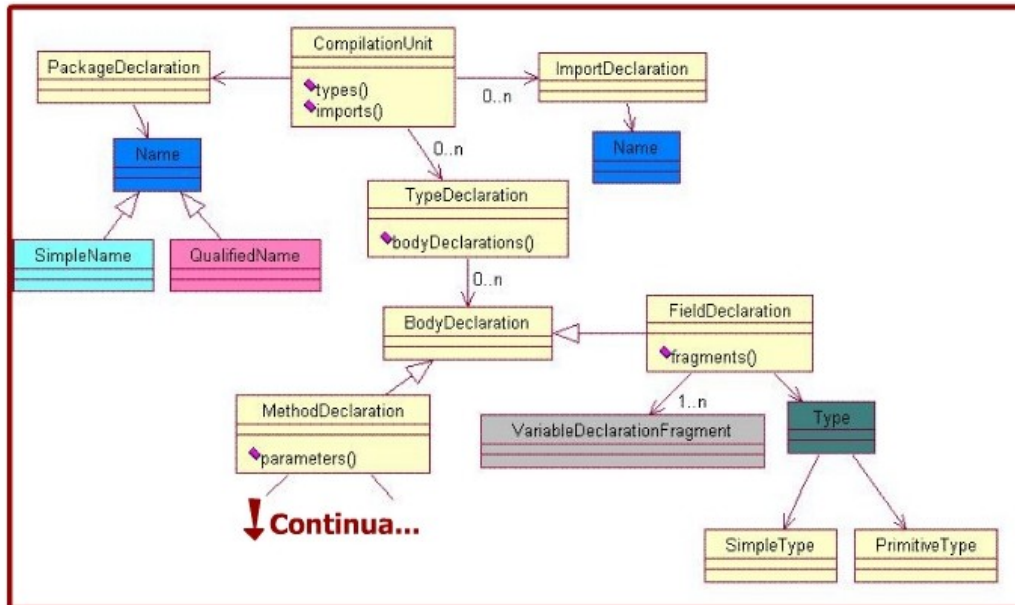


Figura 2.5: primeira parte da AST

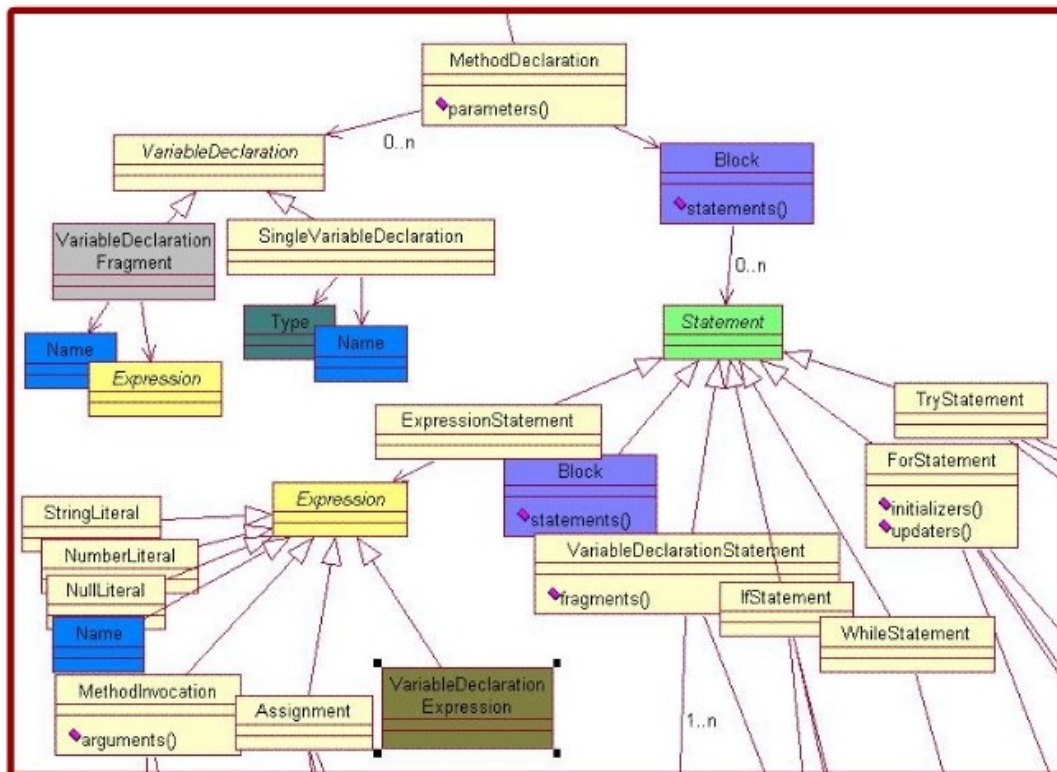


Figura 2.6: segunda parte da AST

Utilizamos o padrão de projeto *visitor* [Gamma, 1995], que veremos em mais detalhes no capítulo de arquitetura, para navegar por essa estrutura. Com isso, podemos fazer checagens de pré-condições de execução dos *refactorings* (técnicas para reestruturar código sem alterar o seu comportamento, visando melhorar características como entendimento do mesmo e facilidade de manutenção [Fowler et al., 1999]) e alterações diretamente na AST, refletidas no código original.

Para obtermos um objeto do tipo *CompilationUnit* (raiz), é preciso executar um método de uma classe de *parser* (*ASTParser*) sobre um documento com código Java™. Um exemplo desse processo pode ser visto a seguir:

```
IDocument doc = new Document("import java.util.List;");
ASTParser parser = ASTParser.newParser(AST.JLS2);
parser.setSource(doc.get().toCharArray());
CompilationUnit cU = (CompilationUnit) parser.createAST(null);
```

Criamos um documento (*doc*) do tipo *IDocument* apenas com uma instrução do tipo *import*. Em seguida, usamos um método da classe *ASTParser* para gerar um objeto do tipo *CompilationUnit* a partir do conteúdo desse documento. Vale salientar que utilizamos o padrão AST.JLS2 para Java 1.4. Se quisermos obter um objeto do tipo *CompilationUnit* para Java 1.5, utilizamos AST.JLS3. Por fim, habilitamos modificações do usuário no nosso objeto do tipo *CompilationUnit*.

```
cU.recordModifications();
```

Para criarmos os nós de uma AST, é preciso obter o objeto AST a partir de um objeto do tipo *CompilationUnit* (raiz), como pode ser visto a seguir:

```
AST ast = cU.getAST();
```

Com esse objeto, é possível criar novos nós para nosso objeto do tipo *CompilationUnit*. A seguir, criamos um novo pacote:

```
PackageDeclaration packageDeclaration = ast.newPackageDeclaration();
packageDeclaration.setName(ast.newSimpleName("exemplo"));
cU.setPackage(packageDeclaration);
```

O objeto da classe AST possui uma série de métodos que permitem criar qualquer objeto do tipo *ASTNode*. Depois de criados, podemos adicioná-los a qualquer outro objeto dessa AST.

Por fim, temos um exemplo de criação de classe:

```
TypeDeclaration type = ast.newTypeDeclaration();
type.setInterface(false);
type.setModifiers(Modifier.PUBLIC);
type.setName(ast.newSimpleName("HelloWorld"));
cU.types().add(type);
```

Criamos um objeto do tipo *TypeDeclaration*, definindo seu nome e propriedades, para adicionarmos à lista de tipos de nossa *CompilationUnit*. Essa lista é necessária porque podemos ter várias classes dentro de um único arquivo .java.

2.2.2 org.eclipse.jdt.core

Esse pacote contém classes que modelam objetos associados à criação, edição e compilação de programas Java™. Alguns desses objetos são encontrados no *workspace* da plataforma. Na Figura 2.7, é possível ver um esquema da relação entre as bibliotecas **org.eclipse.jdt.core** e **org.eclipse.jdt.core.dom**. Em diversos pontos da ferramenta apresentada neste trabalho, precisamos obter ou criar classes dentro de projetos Java™. Necessidades como essa, além de outras de domínio mais específico, justificam a imprescindibilidade de conhecer as estruturas principais dessa biblioteca.

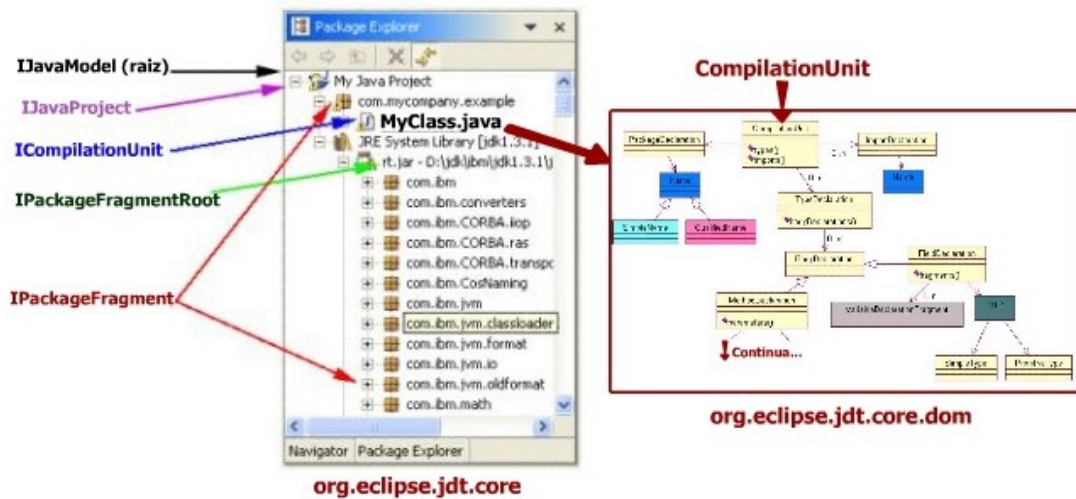


Figura 2.7: obtenção de uma AST a partir de *ICompilationUnit*

Para obtermos nossa AST de um arquivo .java, executamos uma série de passos semelhantes ao processo de obter um objeto do tipo *CompilationUnit* a partir de um documento (descrito no capítulo anterior). A única diferença é a passagem de um objeto que implementa a interface *ICompilationUnit* para o nosso *parser*. Esse objeto pode ser obtido do editor que está aberto no *workbench* ou diretamente de um arquivo de projeto através do *workspace*.

```
ICompilationUnit iCU = ...;
parser.setSource(iCU);
CompilationUnit cU = (CompilationUnit) parser.createAST(null);
cU.recordModifications();
```

2.3 Programação Orientada a Aspectos (POA)

Em vários sistemas desenvolvidos utilizando programação orientada a objetos (OOP), é possível perceber que certos requisitos, em vez de ficarem definidos em apenas um bloco de código ou em uma determinada classe, estão “espalhados” em vários lugares do programa (requisitos transversais). Isso, às vezes, é um fator negativo, porque prejudica o entendimento e a manutenção do código. No exemplo da Figura 2.8, temos uma representação de todas as classes de um projeto. Quanto maior o tamanho da coluna vertical, maior o código da classe. Os trechos em vermelho representam um mesmo requisito. O problema exibido não parece muito

grave. Executando alguns *refactorings*, talvez seja possível resolver esse espalhamento de código sem muita dificuldade.

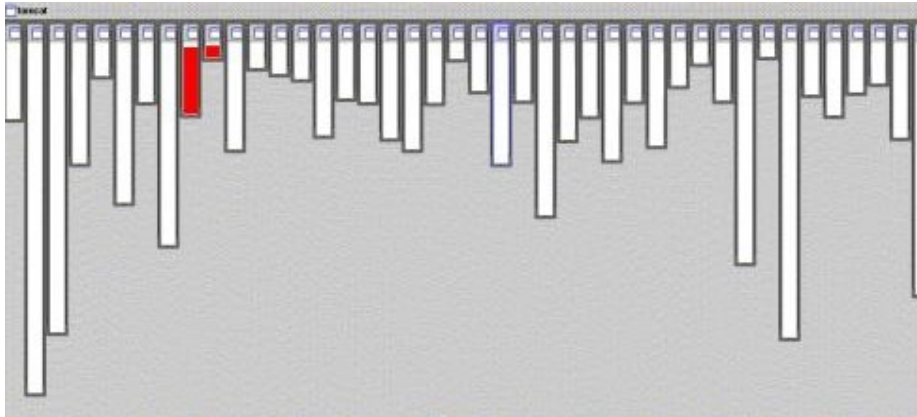


Figura 2.8: espalhamento de código

Não obstante, alguns requisitos não são tão fáceis de isolar. Nos piores casos, a funcionalidade, além de “espalhada”, está “entrelaçada” com código de outros requisitos. Por restrições da linguagem e pela própria arquitetura da implementação, às vezes não é possível modularizar alguns requisitos com recursos de OOP. Um exemplo crítico pode ser visto na Figura 2.9, onde temos o requisito “logar” espalhado pelo programa e entrelaçado com código de outros requisitos.

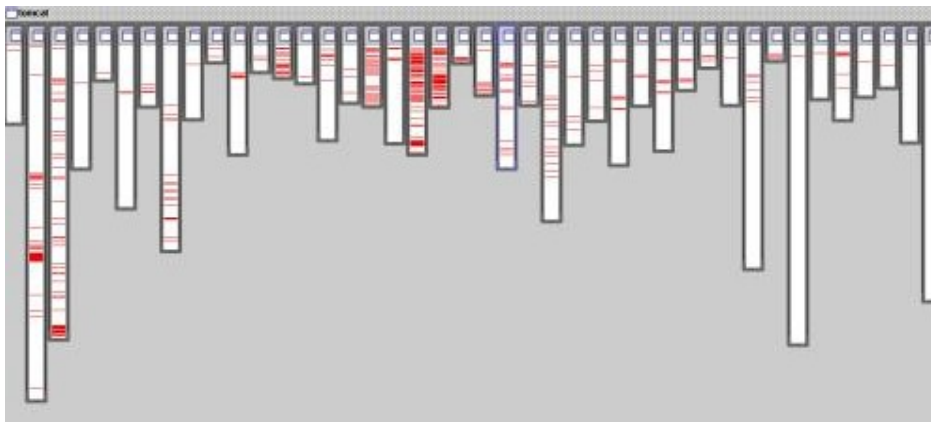


Figura 2.9: espalhamento e entrelaçamento de um requisito

Algumas implicações desses problemas:

- ☑ Redundância: muitos fragmentos de código iguais (ou semelhantes) ocorrem em diversos pontos do código fonte;
- ☑ Forte Acoplamento: os métodos das classes que implementam requisitos transversais geram dependência nos métodos de outras classes;
- ☑ Fraca coesão: métodos contêm instruções que não estão diretamente relacionadas às funcionalidades que eles implementam;
- ☑ Dificuldade de compreensão, manutenção e reutilização: como a implementação do requisito transversal depende do código espalhado pela aplicação, sua compreensão, manutenção e reutilização ficam prejudicadas.

POA [Kiczales, 1997] apresenta uma solução viável para essa problemática. AOP complementa OOP por utilizar um outro tipo de modularidade que coloca a implementação “espalhada” (*crosscutting concerns*) de um requisito transversal dentro de uma simples unidade, chamada aspecto. Essa técnica permite um aumento na coesão e diminui o acoplamento, melhorando a extensibilidade do programa e aumentando o seu reuso.

Um aspecto pode alterar o comportamento de um código base (parte do programa que não é aspecto) aplicando *advice*s (comportamento adicional ou alternativo) sobre um conjunto de *join points* (pontos na estrutura de execução de um programa), capturado por um *pointcut* (descrição lógica desse conjunto). Em muitas linguagens AOP, execução de métodos e referências a atributos são todos exemplos de *join points*. O *pointcut* seria, por exemplo, uma expressão denotando referência para um conjunto particular de atributos.

2.3.1. AspectJ

AspectJ é uma extensão da linguagem Java™ compatível com a mesma, para o domínio de AOP. Ela especifica conceitos e regras de entrelaçamento que definem onde e como o aspecto irá atuar. Esses conceitos serão explanados a seguir:

Join points: pontos bem definidos na execução de um programa, como chamadas e execuções de um método ou construtor, acesso de escrita e leitura de um atributo, execução do tratamento de exceções e inicialização de classes e objetos.

Pointcuts: construção da linguagem para identificar *join points*, filtrando todos aqueles existentes no programa. Têm como principal utilidade expor o contexto de um *join point* para um *advice*, explanado a seguir. Um exemplo de um *pointcut* que captura as chamadas a todos os métodos *setSaldo* do programa é exibido a seguir:

```
pointcut saldoAlterado(): call(void setSaldo(BigDecimal));
```

Advice: especifica o código que será executado quando certos *join points* forem atingidos. Um *join point* pode ser associável a um *advice* através das cláusulas: *before*, *after* e *around*. Um exemplo é exibido a seguir:

```
pointcut saldoAlterado(): call(void Conta.setSaldo(BigDecimal));  
  
before(): saldoAlterado(){  
    <código executado>  
}
```

O *pointcut* *saldoAlterado()* captura todos os *join points* do tipo chamada do método “*void Conta.setSaldo(BigDecimal)*” do programa. O *advice* seguinte (do tipo *before*), determina um trecho de código que deverá ser executado antes de qualquer chamada do método *setSaldo* da classe *Conta*. O *advice* do tipo *before* é muito útil para a checagem de pré-condições.

Por outro lado, o *advice* do tipo *after* é utilizado quando se quer executar um determinado código depois da ocorrência de um *join point* especificado por um

pointcut. Por fim, *advice* do tipo *around*, o *advice* mais expressivo, pode substituir execuções ou inserir contextos nos *join points*.

Inter-type declarations: declara novos membros na classe, como atributos, métodos e construtores. Também permite mudar a hierarquia de tipos, declarando que uma classe implementa novas interfaces ou estende uma nova classe. Um exemplo de inserção de um atributo do tipo boolean e de um método na classe Conta é exibido a seguir:

```
private boolean Conta.ativa = true;
public IType Conta.typeCheck(Environment env) {...}
```

Aspectos: unidade modular de implementação de *crosscutting concerns*. Permite agrupar *pointcuts*, *advices* e *inter-type declarations*. Exemplo:

```
aspect Teste {
    private boolean Conta.ativa = true;
    pointcut saldoAlterado(): call(void Conta.setSaldo(BigDecimal));
    before(): saldoAlterado() { <código executado> }
}
```

2.3.2. AJDT

O *AspectJ Development Tools* (AJDT) é um conjunto de ferramentas para o desenvolvimento de projetos que utilizam *AspectJ* na plataforma Eclipse. Essas ferramentas são integráveis com o JDT do Eclipse SDK, permitindo inclusive a conversão de projetos Java para *AspectJ*.

No nosso projeto, utilizamos algumas bibliotecas do AJDT para manipular aspectos de forma estruturada. Mais detalhes serão vistos no capítulo 4.

3. Projeto da Ferramenta

Neste capítulo, apresentamos o projeto da nossa ferramenta juntamente com uma motivação complementar ao capítulo de introdução, exibindo exemplos concretos de casos de *porting* da indústria. Por fim, apresentamos uma generalização dos *refactorings* que podem ser aplicados a cada um dos tipos de padrão de variação identificados nos estudos dos casos mencionados.

3.1. Problemática

Atualmente, quando uma empresa desenvolve um aplicativo para celular, ela precisa desenvolver várias versões, uma para cada tipo de aparelho, se ela quiser obter maior lucratividade na sua venda. Entretanto, esse processo não é trivial, pois os aparelhos possuem uma série de diferenças que influenciam no código do software: capacidade da memória, poder de processamento, tamanho de tela, dentre outros. Além de que algumas plataformas possuem recursos de programação próprios (API's proprietárias) que não podem ser descartados por maximizar a qualidade da aplicação para seus dispositivos, se utilizados. Como resultado desses fatores, os desenvolvedores são obrigados a produzir dezenas ou até centenas de variações do mesmo jogo. Além disso, a manutenção dessas variações torna-se uma tarefa dispendiosa e mais passível de erro, já que o núcleo funcional comum está disperso nas diferentes variações [Sampaio et al., 2004]. Ou seja, há uma série de características que complicam ainda mais o processo de portar uma aplicação de uma plataforma para outra. Na Figura 3.1, um mesmo jogo é executado em três plataformas diferentes. Para isso acontecer, a implementação precisa ser diferente em cada um desses casos.



64kb, flip 4Mb, flip 100Kb, sem flip

Figura 3.1: aparelhos diferentes, diversas aplicações diferentes

A primeira plataforma, a S40 da Nokia, possui uma restrição de memória em relação às outras plataformas (64Kb). A segunda, a S60 também da Nokia, possui muito mais memória do que as demais, além de possui uma tela maior. A terceira, a T720 da Motorola, possui também pouca memória e uma tela muito pequena, além de não possuir o recurso de *flip*, disponível apenas na API da Nokia, que permite a criação de um “espelhamento” de uma imagem durante a execução do jogo. Esse recurso é de grande utilidade em jogos que invertem imagens constantemente, como o que iremos analisar mais adiante. Esse detalhe evita a necessidade de carregar as imagens invertidas, além das originais. Já no T720, como não temos o recurso de *flip*, é necessário realizar esse procedimento.

A partir da análise de um jogo da indústria em [Sampaio et al., 2004], foi possível identificar os tipos de variação mais freqüentes do código fonte resultante do porte. Apesar da simplicidade do jogo, as diferenças dos aparelhos implicaram num total de 79 modificações. Sendo que o tamanho médio de cada modificação é de aproximadamente 2 linhas, revelando a fina granulação dessas mudanças.

Apesar de todas essas diferenças, muitas partes do código são totalmente iguais, mas, na maioria dos casos, muito difíceis de separar. A Figura 3.2 compara uma mesma classe implementada em duas plataformas diferentes, a T720 e a S60.

```

89  */
90  public void draw(Graphics g) {
91      if(this.isVisible()) {
92  < int offsetX = 0;
93      if (this.collisionCount <=0) {
94          // Draws the dragon
95          g.setClip(this.getX(),
96                  this.getY(),
97                  this.getWidth(),
98                  this.getHeight());
99
100 < if (this.getChSpeed() > 0) {
101 <     if (this.openHounth > 0 || (this.isSpecial && this.isFiring)){
102 <         offsetX = -7*(this.getWidth());
103 <     } else {
104 <         offsetX = -1*(MainCanvas.frame%2 * (this.getWidth()));
105
106         g.drawImage(this.getImage(),
107                    this.getX()+offsetX,
108                    this.getY(),
109                    g.TOP | g.LEFT);
110 <     if (this.openHounth > 0 || (this.isSpecial && this.isFiring)){
111 <         offsetX = 0;
112 <     } else {
113 <         offsetX = this.getImage().getWidth()-this.getWidth() - (MainCanvas.frame%2 *
114 <
115 <         DirectGraphics dg = DirectUtils.getDirectGraphics(g);
116 <         dg.drawImage(this.getImage(),
117 <                    this.getX()-offsetX,
118 <                    this.getY(),
119 <                    g.TOP | g.LEFT,
120 <                    DirectGraphics.FLIP_HORIZONTAL);
121 <     }
122
123     // Draw the breath of fire
124     if (this.isSpecial && this.isFiring) {
125 <         offsetX = -1*(MainCanvas.frame%2 *
126 <         (Resources.dragonFlame.getWidth()/2);
127         g.setClip(this.getX()+this.getWidth(),
128                 this.getY()+this.getHeight(),
129                 Resources.dragonFlame.getWidth(),

```

Figura 3.2: comparação de uma mesma classe para diferentes plataformas

Os trechos em vermelho representam a parte do código que está apenas na plataforma T720. O restante do código é o que temos em comum entre as duas.

Adaptar uma versão de uma plataforma para outra é um trabalho extremamente demorado e complexo, sem falar no aspecto da manutenção, já mencionado. Para amenizar esse impacto, o ideal é separar em módulos as variações específicas de

cada um dos aparelhos da parte em comum entre eles, o núcleo do código. A Programação Orientada a Objetos (POO) é bastante limitada para separar essas características nesse domínio, porque, geralmente, o código de cada uma delas não está definido em um único trecho, mas entrelaçado com outros códigos ou disperso em diversas classes (Figura 3.2).

Na Figura 3.3, identificamos em diagramas de componentes uma série de recursos de determinadas plataformas que estão espalhados por diversas classes. O segundo diagrama, da esquerda para a direita, já possui um componente a mais em relação ao primeiro, além de um recurso novo (círculo verde), com seu código espalhado em mais de uma classe. O último, possui também o mesmo recurso do segundo diagrama (círculo verde), mas dessa vez, ele está espalhado de uma forma diferente. Além dele, mais um novo recurso é acrescentado (quadrado vermelho), além de que dois componentes acabaram sendo substituídos por dois novos (componentes azul e rosa).

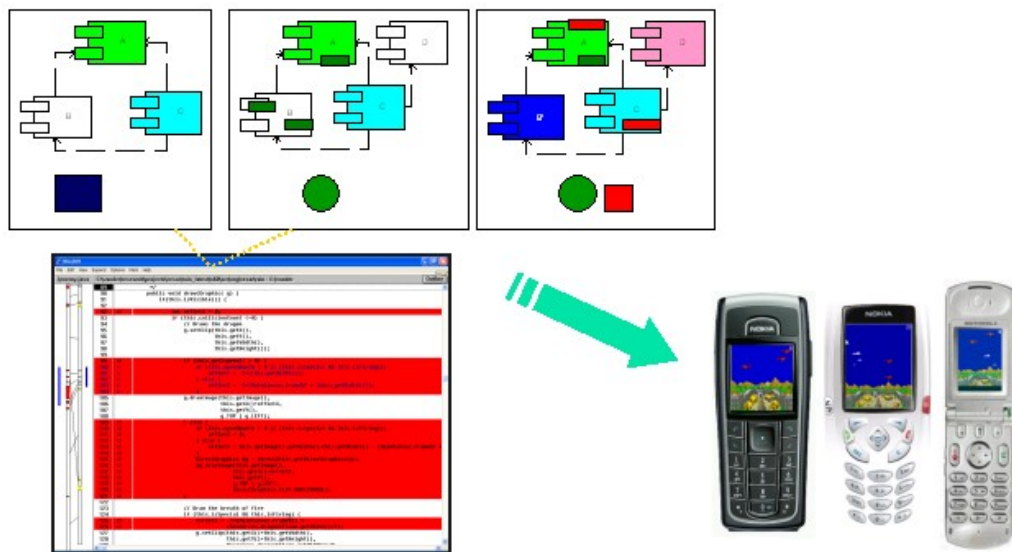


Figura 3.3: pouco reuso e agilidade

3.2. A Ferramenta

A Programação Orientada a Aspectos (POA) pode complementar deficiências da Programação Orientada a Objetos (POO), como vimos no capítulo anterior, em separar requisitos que estão espalhados e entrelaçados pelo código. No nosso escopo, queremos separar características particulares de cada plataforma. A POA também é de grande serventia para esse caso, facilitando o processo de modularização das características dos diferentes aparelhos.

Uma Linha de Produtos de Software representa uma família de produtos em um determinado domínio de aplicação que possuem características em comum. Essa Linha de Produtos é composta por um conjunto de componentes reusáveis e um conjunto de regras de como esses eles são gerenciados. O processo que parte de uma Linha de Produtos e gera um *software* específico é chamado de instanciação. Esse processo consiste em, a partir da composição e adaptação de alguns dos

artefatos reusáveis e do eventual desenvolvimento de novos artefatos, dar origem a um produto específico [Carmo, 2005].

Aliando essas duas técnicas, é possível extrair os pontos de variação desses jogos para aspectos e depois montá-los como uma Linha de Produtos.

Na abordagem da Figura 3.4, a partir de uma implementação inicial para uma plataforma qualquer, queremos obter uma versão para uma outra plataforma. Podemos extrair para aspectos as partes do código que correspondem às características específicas da primeira plataforma que não estão na segunda. Como produto final dessa operação, obtemos um código semelhante às duas plataformas, o qual denominamos de Base, e um conjunto de aspectos correspondentes à primeira plataforma (quadrados azuis), os primeiros artefatos da nossa Linha de Produtos. Para implementarmos as características do nosso segundo produto, editamos cópias dos aspectos do primeiro produto ou criamos novos aspectos para o mesmo (quadrados verdes), com as características que concernem somente à plataforma do segundo produto. Ao final, teremos montado uma Linha de Produtos com dois produtos instanciáveis [Alves et al., 2005].

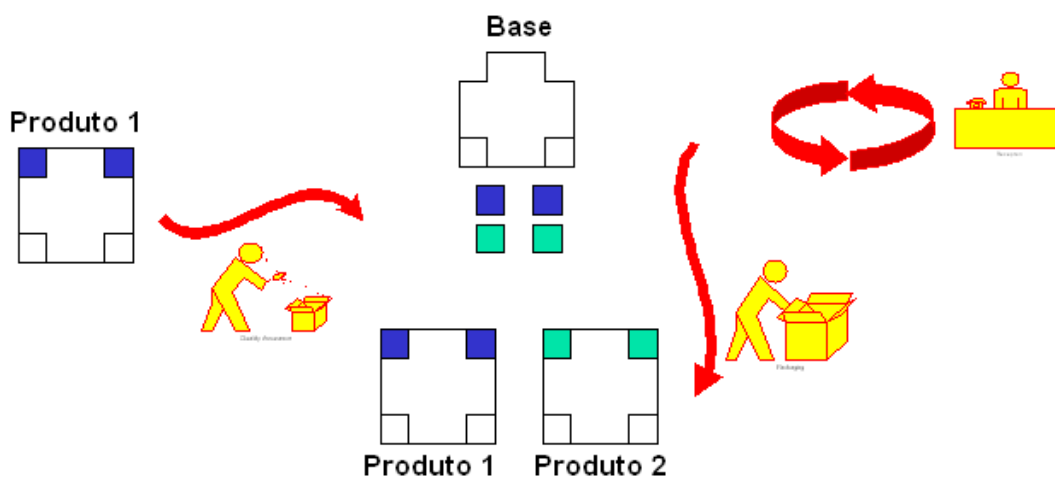


Figura 3.4: Linha de Produtos

Quando quisermos mudar requisitos, precisamos apenas alterar o código da base ou um aspecto referente a um determinado produto. Quando quisermos instanciar um novo produto, simplesmente separamos o código Base e os aspectos de um determinado produto. Se quisermos adicionar um novo produto à Linha de Produtos, precisamos apenas implementar novos aspectos que atuam sobre o código base, que podem até mesmo ser cópias alteradas das já existentes. Algumas vezes, teremos que alterar o código da base por causa da chegada de um novo produto, gerando novos aspectos que serão utilizados por mais de um produto já existente na Linha. Esse processo tende a se tornar menos freqüente, à medida que formos adicionando mais novos produtos ao nosso conjunto, pois iremos obter uma Base cada vez mais genérica.

Nossa ferramenta é uma versão inicial de um ambiente que permitirá a criação de uma Linha de Produtos para jogos em Java™ utilizando aspectos de *AspectJ*. Ela é

implementada como um *plug-in* do Eclipse, tendo como objetivo neste trabalho dispor operações de *refactoring* para a extração de pontos de variação para aspectos. Essas operações extraem padrões semelhantes de variações identificadas em jogos já portados pela indústria [Alves et al., 2005]. Os padrões que extraímos nessa implementação podem ser vistos a seguir.

3.3. Refactorings

Estratégias são definidas para cada processo de extração de variações de código para aspectos. Cada uma delas define quais *refactorings* devem ser aplicados a determinados trechos de código que seguem um padrão, de forma que todos os pontos relativos às variações sejam extraídos.

Com base na análise já mencionada de código de jogos já portados na indústria, [Alves et al., 2005] definiu uma série de *refactorings* que podem ser aplicados no código para extrair as variações características de cada uma das plataformas envolvidas no processo. Transcrevemos a seguir todos aqueles que foram implementados na nossa ferramenta, juntamente com suas pré-condições de execução. Acrescentamos também motivações de utilização para cada um deles, baseadas em experiências reportadas pela equipe que desenvolveu esse trabalho de identificação de padrões. A motivação será um exemplo de aplicação para um desses jogos estudados.

3.3.1 Refactoring Extract Method to Aspect

Esse *refactoring* é útil para extrairmos variações que estejam no meio do corpo de um método ou construtor. Ele extrai o código para um *inter-type declaration* de declaração de método de um aspecto e deixa uma chamada para esse método no lugar do código extraído. A Figura 3.5 generaliza esse procedimento.

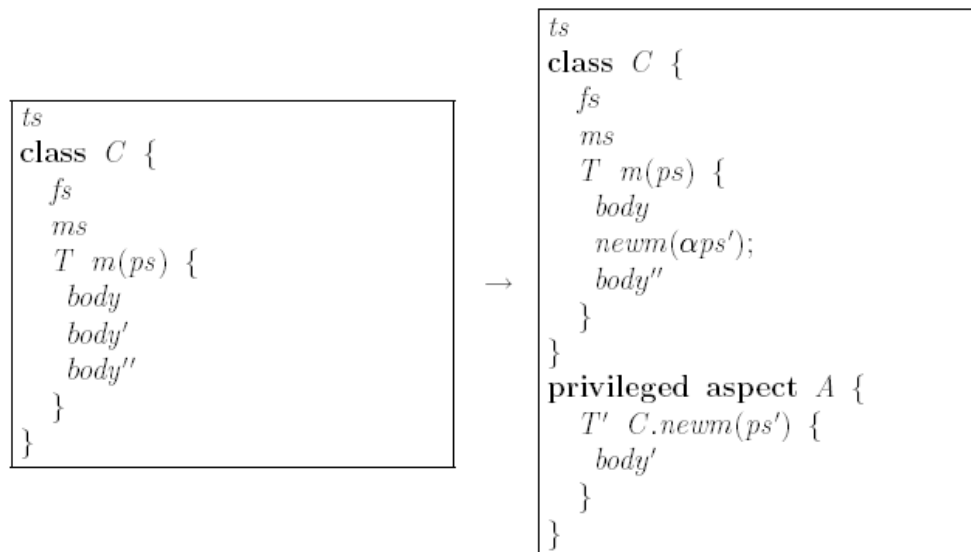


Figura 3.5: Refactoring Extract Method to Aspect

O código *body* de um método ou construtor *m* é extraído para um aspecto, atendendo à seguinte pré-condição:

- ☑ *body* não modifica mais do que uma variável local que seja usada em *body*”.

No lugar do código extraído é colocada uma chamada de método para uma declaração de método de nome *newm* que será criada como *inter-type declaration* dentro do aspecto.

Exemplo de aplicação

Utilizando o recurso de *flip*, existente em plataformas como S40 e S60, conseguimos diminuir o tamanho de alguns métodos. Um dos métodos do jogo analisado é responsável por desenhar os estados de duas catapultas. Uma delas é o espelhamento da outra (Figura 3.6).



Figura 3.6: catapultas

Precisamos apenas desenhar a primeira catapulta; a segunda, é obtida utilizando o recurso de *flip* nos procedimentos de desenho primeira. Em contrapartida, esse recurso não está disponível no T720, fazendo com que tenhamos que desenhar todos os estados novamente para a segunda catapulta. Essa segunda operação é um ponto de variação que pode ser extraído com esse *refactoring*. No aspecto referente às plataformas S40 e S60, teremos um *inter-type declaration* que faz a operação de *flip*; no aspecto da plataforma T720, teremos a nova operação de desenho. Esse exemplo pode ser visto, de forma resumida, no capítulo 4.

3.3.2 Refactoring Extract Resource to Aspect - after

Extraí uma atribuição de um atributo da classe para o aspecto, juntamente com sua declaração. No aspecto, a nova atribuição vira um *inter-type declaration* de declaração de atributo e a atribuição é inserida dentro de um *advice*. Essa atribuição pode estar dentro de um método ou construtor. O procedimento pode ser aplicado também a uma lista de atribuições consecutivas. A Figura 3.7 ilustra esse processo.

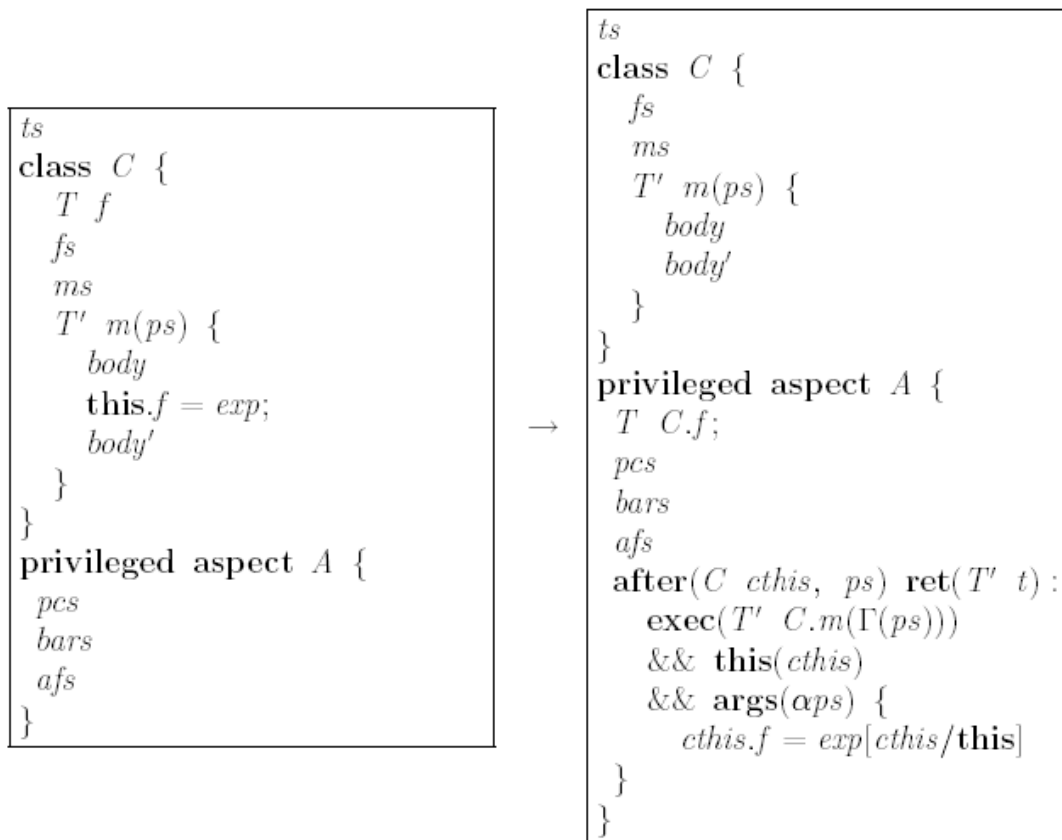


Figura 3.7: Refactoring Extract Resource to Aspect - after

A atribuição *this.f = exp* é extraída juntamente com sua declaração *T f* para o aspecto *A*. Essa operação atende às seguintes pré-condições:

- exp* não utiliza variáveis locais;
- exp* não chama construções do tipo *super*;
- O atributo *f* não é utilizado em nenhum outro lugar da classe;
- O aspecto *A* tem a maior precedência em *join points* envolvendo a assinatura do método ou construtor que contém a atribuição selecionada.

Exemplo de aplicação

Em plataformas que não possuem o recurso de *flip*, como o T720, é necessário carregar as imagens invertidas das originais que precisam ser invertidas durante a execução do jogo (Figura 3.8).



Figura 3.8: imagens invertidas

O carregamento de imagens é feito através de atribuições. Nesse caso, devemos extrair as atribuições referentes às imagens invertidas para um aspecto do T720. Esse exemplo pode ser visto, de forma resumida, no capítulo 6.

3.3.3 Refactoring Extract Context

Extrai o contexto de um bloco de código selecionado. Esse trecho selecionado deve fazer parte do corpo de um método ou de um construtor e, além disso, ele deve estar inserido, e somente ele, dentro do corpo de alguma estrutura de controle (ex. laços *if* e *while* e blocos *try-catch*). Essa estrutura de controle pode estar inserida também em outra estrutura de controle. Esse conjunto de estruturas de controle representa o contexto que será extraído. Nenhuma outra estrutura pertencente ao método pode estar fora desse bloco de estruturas de controle aninhadas. Se existir alguma, podemos aplicar o *refactoring Extract Method* do JDT de forma a separar o código dela. Vale salientar que o código que está inserido em cláusulas *else* de estruturas de controle *if-else* que contém o código selecionado também será extraído juntamente com sua estrutura. A Figura 3.9 ilustra esse processo.

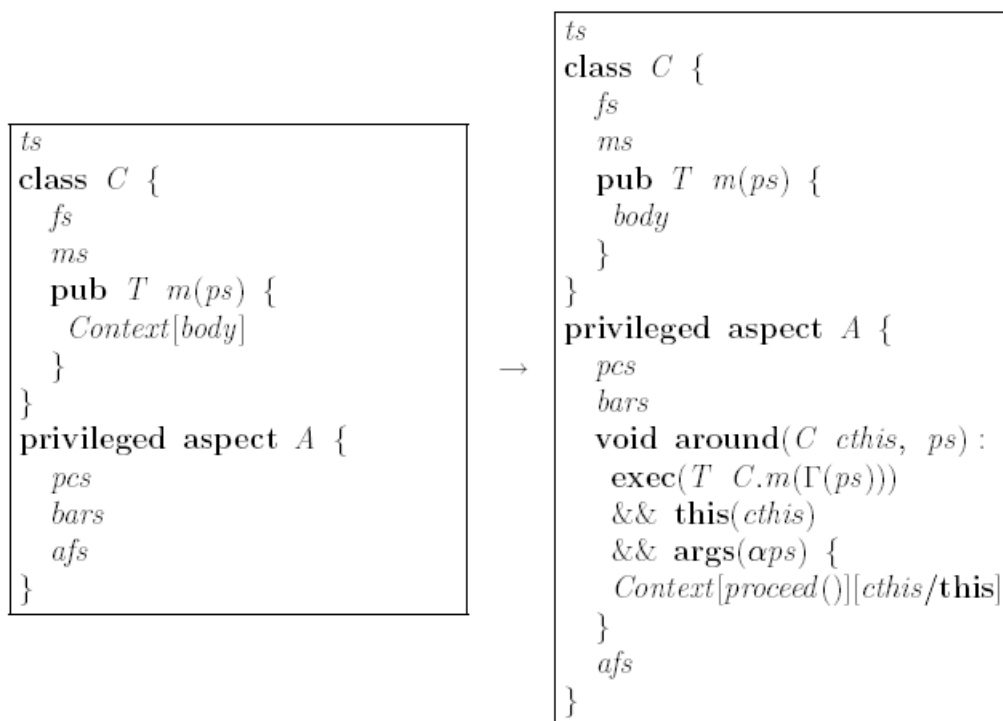


Figura 3.9: Refactoring Extract Context

O contexto que envolve *body* deve ser extraído para um aspecto A. As seguintes pré-condições devem ser garantidas:

- O contexto de *body* não chama construções do tipo *super* e *return*;
- body* não usa variáveis locais declaradas no contexto;
- Não há aspecto afetando o método que contém o contexto.

Exemplo de aplicação

Em plataformas que possuem *flip*, é normal termos estruturas de controle que envolvem operações de desenho. Essas estruturas definem, de acordo com o estado atual de um objeto da tela, quais operações de desenho devem ser efetuadas sobre esse objeto. Operações como fazer um dragão com imagem invertida cuspir fogo com imagem invertida, por exemplo. Em plataformas sem *flip*,

essas estruturas podem ser removidas e outras novas operações devem ser adicionadas em outros blocos, possivelmente usando a operação de *Extract Method to Aspect*. Combinado com esse último *refactoring* e com outras operações simples, é possível extrair esses contextos eficientemente para aspectos. Esse exemplo pode ser visto, de forma resumida, no capítulo 6.

3.3.4 Refactoring Extract Before Block

Extrai o primeiro bloco de código de um método ou construtor. A Figura 3.10 ilustra esse processo.

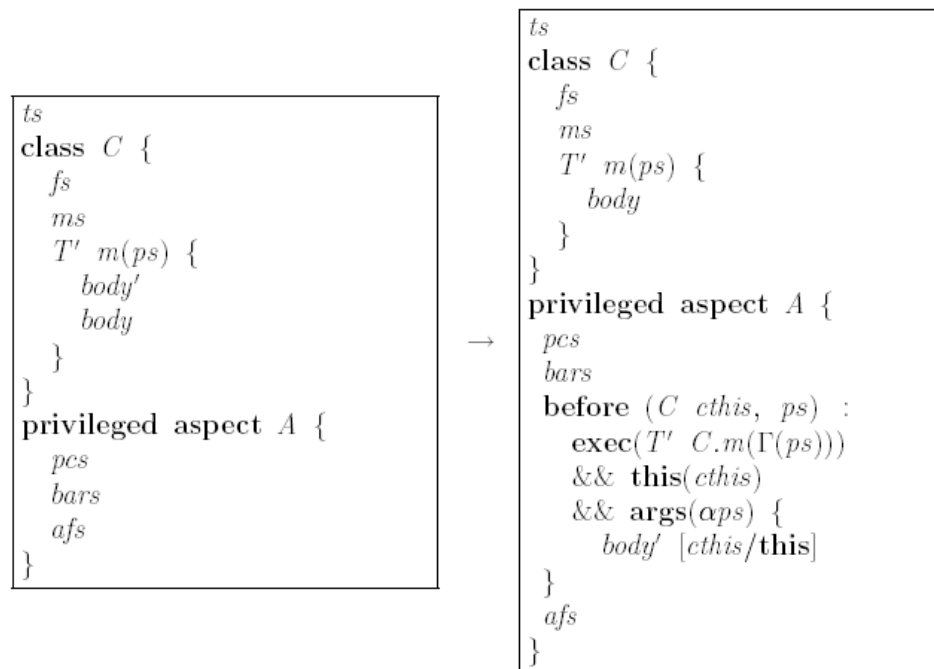


Figura 3.10: Refactoring Extract Before Block

O bloco *body'* deve ser extraído para o aspecto A. As seguintes pré-condições devem ser verdadeiras:

- ☑ *body'* é o primeiro bloco de código do método ou construtor *m*;
- ☑ *body* não usa variáveis locais declaradas em *body'*;
- ☑ *body'* não chama construções do tipo *super* e *return*;
- ☑ O aspecto A tem a menor precedência em *join points* envolvendo a assinatura do método ou construtor que contém o código que será extraído.

Exemplo de aplicação

O aparelho da plataforma S60 possui uma quantidade maior de memória do que os aparelhos da S40 e da T720. Por causa disso, ela pode carregar toda as imagens que vão ser utilizadas na memória antes do jogo iniciar. Nas outras duas, essas imagens devem ser carregadas sob demanda, ou seja, apenas no momento em que elas forem utilizadas.

Quando queremos desenhar algo na tela, chamamos um método específico para isso. No S40 e no T720, por exemplo, esse método irá, antes de iniciar a operação, decidir quais imagens devem ser carregadas. Esse bloco de inicialização sob demanda pode ser extraído para um aspecto que será compartilhado pelo S40 e pelo T720. Esse exemplo pode ser visto, de forma resumida, no capítulo 6.

3.3.5 Refactoring Extract After Block

Extrai o último bloco de código de um método ou construtor. A Figura 3.11 ilustra esse processo.

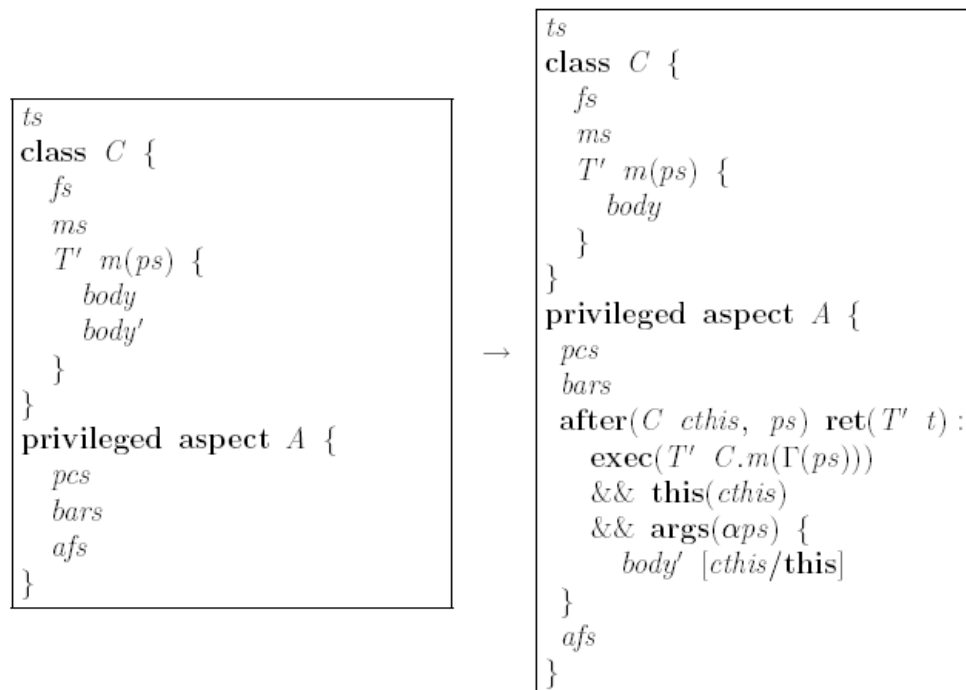


Figura 3.11: Refactoring Extract After Block

O bloco *body'* deve ser extraído para o aspecto A. As seguintes pré-condições devem ser estabelecidas:

- body'* é o último bloco de código do método ou construtor m;
- body'* não usa variáveis locais declaradas em *body*;
- body'* não chama construções do tipo *super*;
- O aspecto A tem a maior precedência em *join points* envolvendo a assinatura do método ou construtor que contém o código que será extraído.

Exemplo de aplicação

Ainda não há aplicação para esse *refactoring*. Ele foi desenvolvido para complementar o *refactoring* anterior, o *Extract Before Block*, aproveitando boa parte de sua implementação.

3.3.6 Refactoring Change Class Hierarchy

Esse *refactoring* altera a superclasse de uma classe para a próxima superclasse da hierarquia. A Figura 3.12 ilustra esse processo.

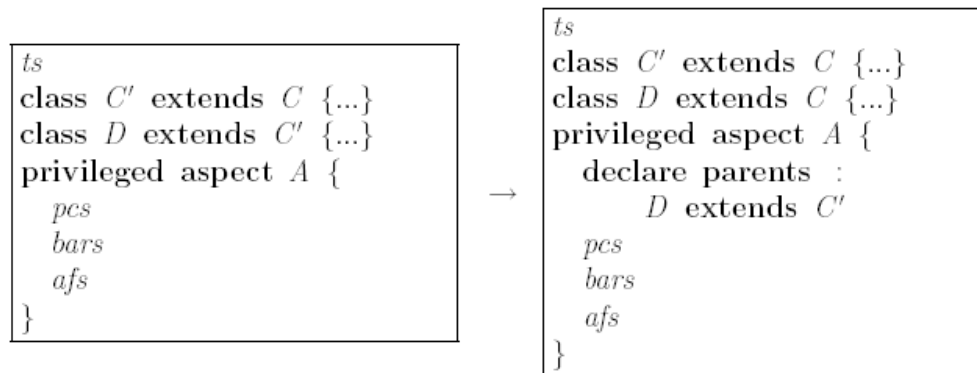


Figura 3.12: Refactoring Change Class Hierarchy

O esquema da Figura 3.13 facilita o entendimento desse processo.

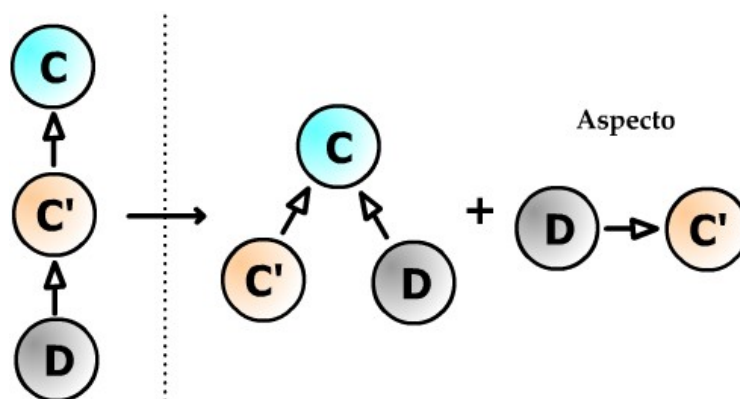


Figura 3.13: processo do *refactoring* da classe

Exemplo de aplicação

Nas plataformas S40 e S60, a classe *MainCanvas* é subclasse de *FullCanvas*. No T720, *MainCanvas* é subclasse de *Canvas*, que é superclasse de *FullCanvas*. *FullCanvas* é uma classe da API da Nokia. Então, com esse *refactoring*, alteramos a declaração “*extends FullCanvas*” para “*extends Canvas*” e criamos um aspecto para as plataformas da Nokia que substituem *Canvas* por *FullCanvas*, na hora em que criarmos um produto para essas plataformas. Esse exemplo pode ser visto, de forma resumida, no capítulo 6.

4. Arquitetura da Ferramenta

A ferramenta apresentada neste trabalho foi desenvolvida com a intenção de facilitar a extração de pontos de variação de um sistema já desenvolvido para aspectos. Nela, o usuário terá que selecionar trechos de código que devem ser extraídos de cada classe e executar um dos *refactorings* pré-definidos. Para cada trecho de código há um *refactoring* diferente, como foi visto no capítulo anterior. O desenvolvedor deve, antes de efetuar as extrações, elaborar uma estratégia de quais *refactorings* ele vai usar e onde ele irá aplicá-los.

Uma perspectiva da ferramenta, denominada *Product Lines*, exibe uma série de botões (ações) na barra de ferramentas e na barra de menu do *workbench*, que executam os *refactorings* explanados no capítulo anterior (Figura 4.1).

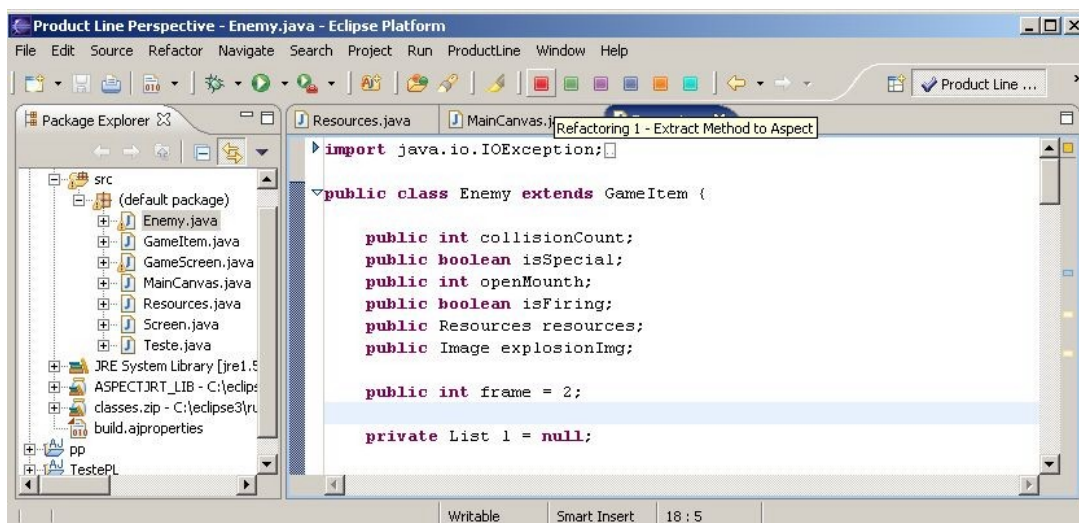


Figura 4.1: ferramenta de ProductLines

A seguir, iremos discorrer sobre a arquitetura do *Refactoring Extract Method To Aspect*. A estrutura dos outros *refactorings* é bastante similar a essa. As principais diferenças estão nas verificações das pré-condições, na geração do código do aspecto e na alteração da classe de origem. Mais detalhes dos outros *refactorings* serão vistos no capítulo referente à implementação.

4.1. Refactoring Extract Method To Aspect

O *Refactoring Extract Method To Aspect* funciona de acordo com o seguinte fluxo de eventos:

1. O usuário seleciona um trecho de código de um arquivo Java™ aberto em um editor e clica no botão *Refactoring Extract Method To Aspect*. Esse trecho de código deve corresponder a uma seleção que faça parte do corpo de um método ou de um construtor (Figura 4.2);

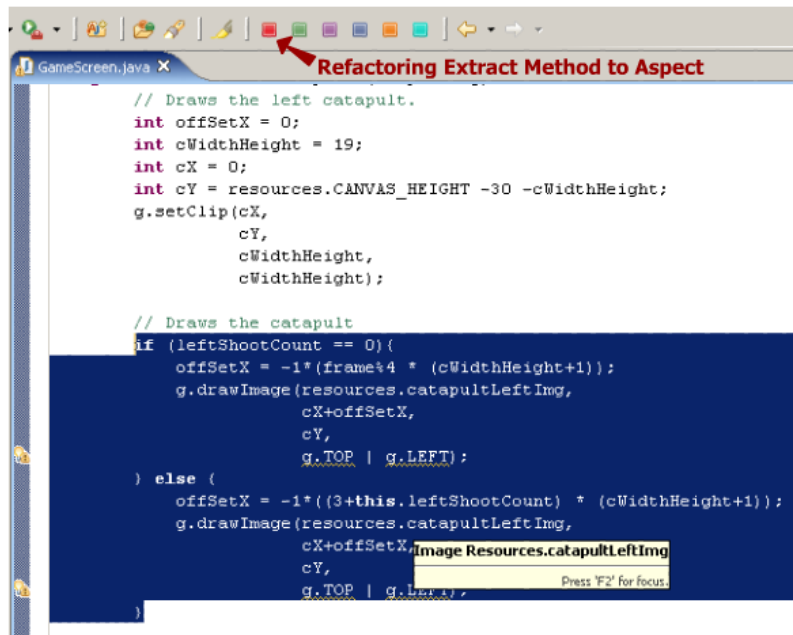


Figura 4.2: seleção de trecho de código

2. Uma caixa de diálogo é aberta, perguntando se o usuário quer gerar o código correspondente ao *refactoring* em um aspecto já existente ou se quer gerar em um novo aspecto. O nome do método que será criado no aspecto como um *inter-type declaration* (ponto de variação) e que irá guardar o código selecionado também é solicitado nessa janela (Figura 4.3);

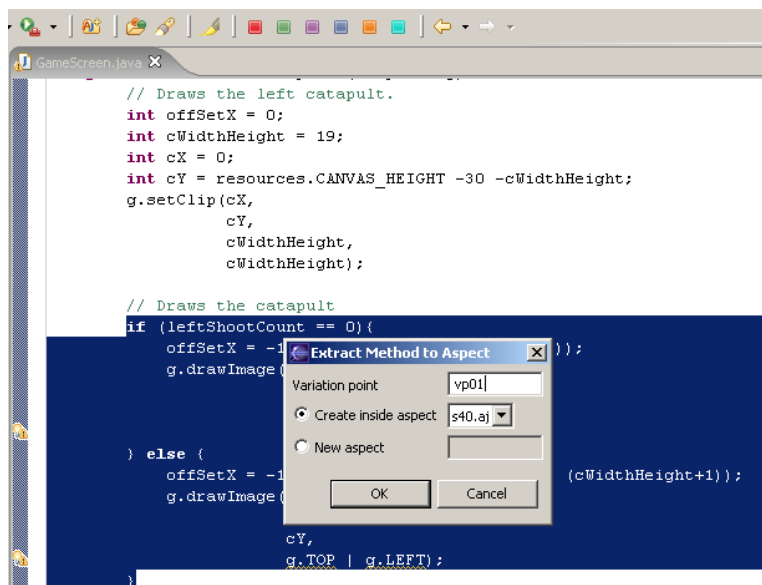


Figura 4.3: seleção de opção

3. A verificação das pré-condições no código selecionado com o objetivo de garantir que o código gerado vai estar sintaticamente e semanticamente correto é executada. A classe é alterada (Figura 4.4);

```

public void keyPressed(int keyCode, int keyAction) {
}

public void paint(Graphics g) {
// ....
    this.drawCatapults(g);
// ....
}

private void drawCatapults(Graphics g) {
// Draws the left catapult.
    int offSetX = 0;
    int cWidthHeight = 19;
    int cX = 0;
    int cY = resources.CANVAS_HEIGHT - 30 - cWidthHeight;
    g.setClip(cX,
              cY,
              cWidthHeight,
              cWidthHeight);

    offSetX = vp01(g, cWidthHeight, cX, cY);
}

```

Figura 4.4: arquivo .java alterado

4. O código em *AspectJ* é gerado e escrito em um aspecto já existente ou em um novo aspecto, dentro do mesmo pacote do arquivo .java, conforme a opção escolhida no passo 2 (Figura 4.5).

```

import javax.microedition.lcdui.Graphics;

privileged aspect s40 {

    public int GameScreen.vp01(Graphics g, int cWidthHeight, int cX, int cY) {
        int offSetX;

        if (leftShootCount == 0) {
            offSetX = -1*(frame%4 * (cWidthHeight+1));
            g.drawImage(resources.catapultLeftImg,
                       cX+offSetX,
                       cY,
                       g.TOP | g.LEFT);
        } else {
            offSetX = -1*((3+this.leftShootCount) * (cWidthHeight+1));
            g.drawImage(resources.catapultLeftImg,
                       cX+offSetX,
                       cY,
                       g.TOP | g.LEFT);
        }
        return offSetX;
    }
}

```

Figura 4.5: código gerado

4.1.1 Padrão *visitor* do JDT

Antes de discorrermos sobre a arquitetura do *refactoring*, é importante introduzir o padrão de projeto *visitor* adotado pelo JDT, utilizado para tornar possível a navegação da estrutura da AST do código .java analisado, com o objetivo de verificar pré-condições ou obter referências de nós específicos. Esse padrão pode ser visto detalhadamente em [Gamma, 1995].

Qualquer classe subtipo de *ASTNode*, do pacote ***org.eclipse.jdt.core.dom***, representa um nó de AST. A classe *ASTNode*, e conseqüentemente todos os nós de uma AST, possui um método *accept(ASTVisitor)*, responsável por receber um objeto do tipo *ASTVisitor*. Esse objeto tem a função de *visitor*: “navegar” por toda a estrutura da AST, podendo analisar seus detalhes. De fato, não é o *visitor* que “navega” pela AST. Os métodos *accept* de cada nó são os responsáveis por enviar o *visitor* como argumento para todos os métodos *accept* dos seus filhos. São eles que fazem, realmente, o *visitor* percorrer toda a estrutura da AST. Recursivamente, essa estratégia resulta em uma “navegação” completa por toda a árvore, a partir do nó que executou a primeira chamada ao método *accept*. Na Figura 4.6, temos a ordem dos passos de navegação do *visitor* a partir da raiz da AST do arquivo .java.

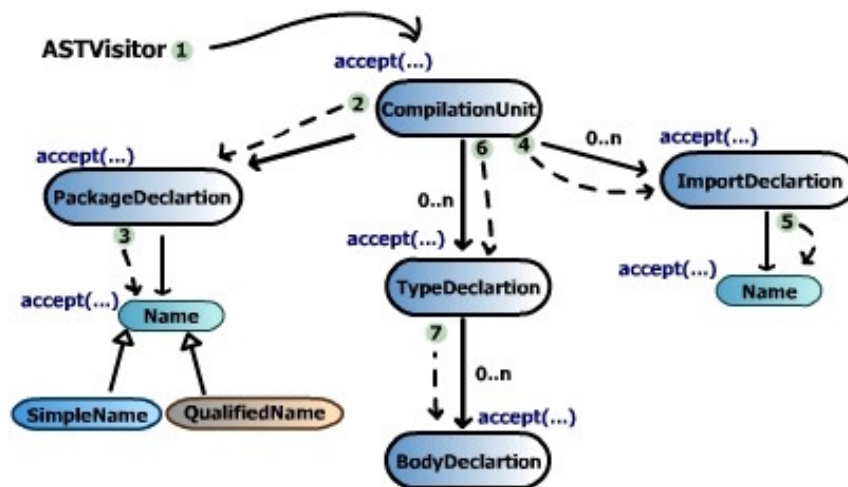


Figura 4.6: padrão *visitor*

O usuário desse recurso precisa apenas criar seu próprio *visitor* como subclasse da classe básica *ASTVisitor*. Depois de instanciá-lo, é necessário apenas passá-lo como argumento para qualquer método *accept* de um *ASTNode* para que a navegação seja efetuada.

Qualquer *visitor* sempre irá percorrer todas as estruturas da AST. O único detalhe que o desenvolvedor precisa acrescentar é dizer quais nós deverão ser analisados. Ele faz isso re-implementando os métodos *visit(...)* e *endVisit(...)* para cada nó que ele quiser analisar. O *visit* realiza a checagem de pré-condições que um nó precisa ter para que o método *endVisit* seja executado; o *endVisit*, executa a ação. Por exemplo, para um *visitor* coletar uma lista de todos os nós do tipo *Name* de uma AST, é necessário apenas implementar o seguinte código:

```
public MyVisitor extends ASTVisitor {
    public List list = new ArrayList();
    ...
    public void endVisit(Name node) {
        list.add(node);
    }
}
```

Depois, instanciar o *visitor* e passá-lo como argumento para um método *accept* de um objeto do tipo *ASTNode*.

```
CompilationUnit cU = ...;
```

```
MyVisitor myVisitor = new MyVisitor();
cU.accept(myVisitor) ;
```

Depois da execução do método *accept*, toda a AST já terá sido percorrida. Já é possível obter a lista de objetos do tipo *Name* do *visitor*.

```
List list = myVisitor.list;
```

O *visitor* é um padrão muito usado no nosso projeto. Seu entendimento é essencial para a compreensão das próximas etapas.

4.1.2 Arquitetura do *plug-in*

As principais classes do projeto estão divididas nos seguintes pacotes:

- ☑ **productLine.actions**: contém todas as ações dos *refactorings* do *plug-in*;
- ☑ **productLine.core.refactoring1**: contém as classes principais do *Refactoring Extract Method to Aspect*. Demais *refactorings* possuem um pacote equivalente;
- ☑ **productLine.core.util.checkers**: contém um conjunto de métodos que verificam pré-condições. É utilizado por quase todos os *refactorings*;
- ☑ **productLine.core.util**: possui *visitors* que são utilizados por mais de um *refactoring*, além de classes com métodos de uso comum entre eles.

Os pacotes do projeto podem ser vistos na Figura 4.7. Demais pacotes não serão discutidos por terem menor importância para o trabalho.



Figura 4.7: pacotes do *plug-in*

Antes de iniciar a implementação de qualquer *plug-in*, é necessário definir suas extensões no arquivo *manifest*. Para as ações, é necessário apenas criar uma extensão para o ponto de extensão **org.eclipse.ui.actionSets**, definindo a ação que será criada.

```
<extension point="org.eclipse.ui.actionSets">
...
  <action
    label="Refactoring 1 - Extract Method to Aspect"
    class="productLine.actions.Refactoring1ActionDelegate"
    ...>
  </action>
```

Toda extensão, quando criada, precisa ter uma classe associada que implementa a interface do ponto de extensão usado, como foi visto no capítulo 2. Essa classe é a *Refactoring1ActionDelegate*. Ela tem a função de executar um evento quando o

botão correspondente ao *refactoring* é clicado pelo usuário. Um esquema desse relacionamento pode ser visto na Figura 4.8.

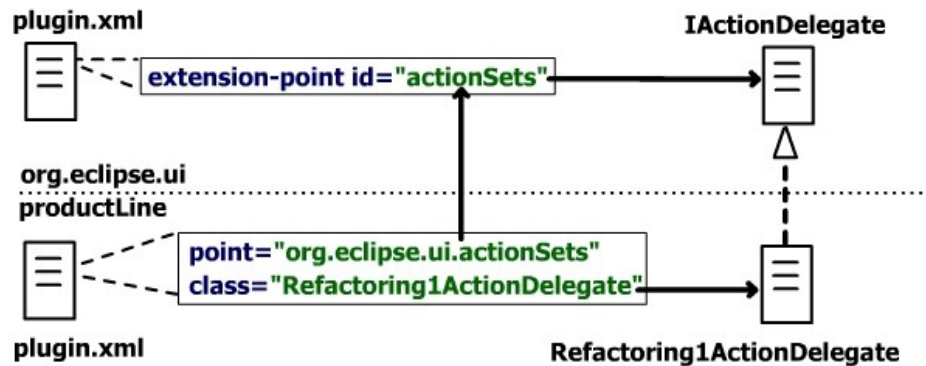


Figura 4.8: relacionamento da extensão com o ponto de extensão

Quem invoca a ação é o *plug-in* do pacote **org.eclipse.ui**. O *plug-in* do pacote **productLine** tem apenas a função de registrar a ação.

A classe *Refactoring1ActionDelegate* precisa implementar um conjunto de métodos definido pela interface que ela implementa. Todos esses métodos serão explicados a seguir.

O método *run(IAction)* é o responsável por executar a ação. Mas, antes dele ser iniciado, o método *init(IWorkbenchWindow)* é chamado. Esse método *init* é de grande serventia por receber como argumento da plataforma um objeto que implementa a interface *IWorkbenchWindow*.

```
private IWorkbenchWindow window;
...
public void init(IWorkbenchWindow window) {
    window = window;
}
```

O objeto “window” representa a janela do *workbench* do Eclipse, ou seja, através dele podemos acessar qualquer estrutura da tela atual da nossa aplicação. No nosso caso, queremos acessar o editor que está ativo no momento da chamada da ação. Esse editor deve ter aberto um arquivo .java, onde queremos aplicar nosso *refactoring*, restrito ao domínio de aplicações implementadas na linguagem Java™. Isso pode ser feito da seguinte maneira:

```
IWorkbenchPage iWorkbenchPage = window.getActivePage();
IEditorPart iEditorPart = iWorkbenchPage.getActiveEditor();
```

Através da janela, obtemos a página ativa, que corresponde ao editor e ao conjunto de visões abertas. Da página (*IWorkbenchPage*), acessamos o editor que está ativo. Em seguida, obtemos o conteúdo desse editor com:

```
IEditorInput editorInput = iEditorPart.getEditorInput();
```

Por fim, com o auxílio de um objeto que implementa *IWorkingCopyManager*, conseguimos obter outro objeto que implementa a interface *ICompilationUnit* a partir do conteúdo do editor.

```
IWorkingCopyManager copyManager = JavaUI.getWorkingCopyManager();
ICompilationUnit iCU = copyManager.getWorkingCopy(editorInput);
```

JavaUI é uma classe estática do JDT que possui um conjunto de métodos que servem para obter dados especiais do *workbench*.

Com um objeto que implementa a interface *ICompilationUnit*, já é possível alterar a estrutura da árvore sintática do nosso arquivo *.java*, como vimos no capítulo 2. Na Figura 4.9, um diagrama de classes simplificado mostra as classes que estão relacionadas com *Refactoring1ActionDelegate*.

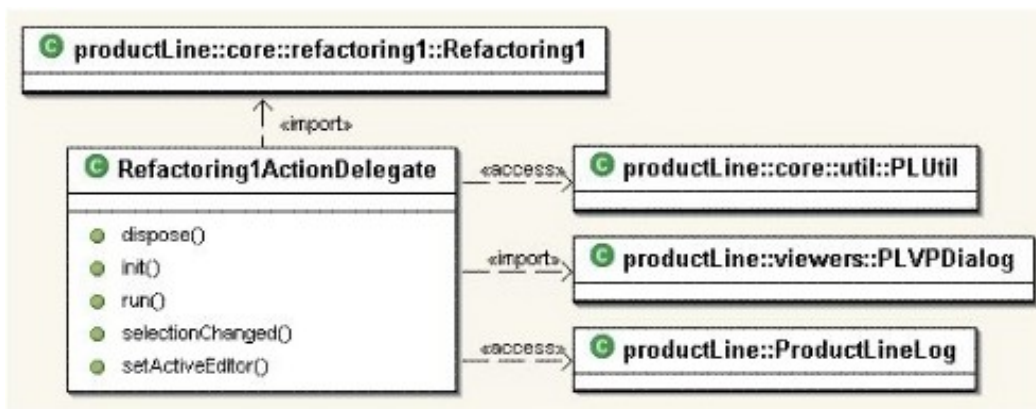


Figura 4.9: Refactoring1ActionDelegate e classes associadas

Além do objeto que representa o arquivo *.java*, precisamos obter a seleção de código onde será aplicado o *Refactoring Extract Method to Aspect*. O método *selectionChanged(IAction, ISelection)* tem a função de capturar qualquer texto ou estrutura (ex. arquivos) que esteja selecionada no *workbench*. Nesse caso, restringimos a seleção apenas a texto.

```
private ITextSelection text;

public void selectionChanged(IAction action, ISelection selection) {
    if(selection instanceof ITextSelection) {
        text = (ITextSelection) selection;
    }
}
```

Com o objeto referenciado pela variável “*text*”, completamos o conjunto necessário para dar início ao procedimento de *refactoring*.

O método *run()* é o próximo a ser executado. Ele tem a função de solicitar informações do usuário como o nome do aspecto que receberá o conteúdo da operação e o nome do ponto de variação que será criado no lugar do código selecionado. Mais detalhes desse método estão no próximo capítulo.

4.1.3 Processo de Refactoring

O próximo passo é checar as pré-condições de validade da seleção de código e obter uma lista de estruturas no formato das classes do pacote **org.eclipse.core.dom** a partir dessa seleção. Através dessas estruturas, podemos realizar operações de verificação e alteração da AST e, conseqüentemente, do código fonte. As classes responsáveis por essa tarefa podem ser vistas na Figura 4.10.

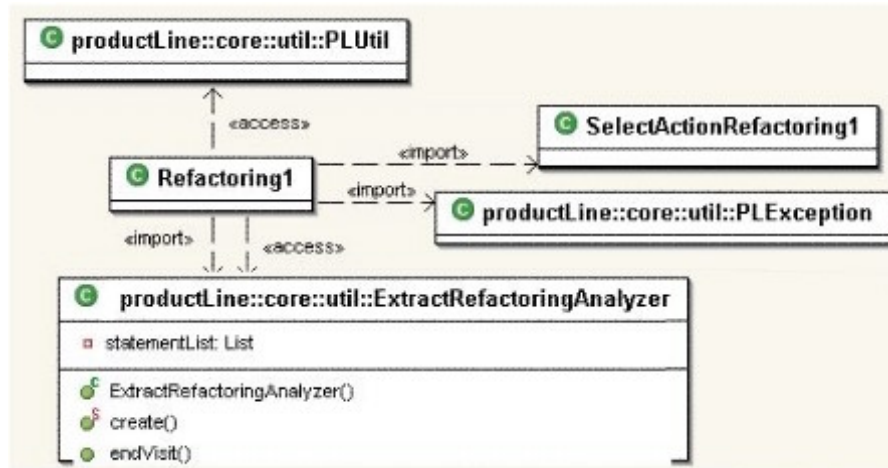


Figura 4.10: Refactoring1 e classes associadas

Um objeto do tipo *Refactoring1*, instanciado pelo método *run* do passo anterior, executa o processo de obtenção de um objeto do tipo *CompilationUnit* a partir do objeto que implementa a interface *ICompilationUnit*, passado como argumento por *run*. Esse processo é delegado à classe **PLUtil**, mas pode ser visto com mais detalhes no capítulo 2.

Com o objeto do tipo *CompilationUnit*, é possível fazer a navegação da AST do código .java utilizando o padrão *visitor* (capítulo 4.1.1). Utilizaremos esse recurso para validar a seleção de código do usuário e depois obter uma lista de elementos estruturados no formato das classes usadas no pacote **org.eclipse.jdt.core.dom**.

O JDT também implementa uma série de subclasses da classe *ASTVisitor*. Uma delas é a *SelectionAnalyzer*, que analisa um objeto do tipo *Selection* (seleção do usuário, facilmente obtido com nosso objeto que representa o texto selecionado) sobre um objeto do tipo *ASTNode* e seus filhos (nós abaixo da sua hierarquia). Ela verifica a validade da seleção do usuário (limite da seleção do texto e *tokens* selecionados incompletos) percorrendo a estrutura abaixo do objeto do tipo *ASTNode* à procura dos elementos que estão no objeto do tipo *Selection*. Se encontrar, ele executa as verificações e adiciona os objetos associados a uma lista.

Também há implementações de subclasses de *SelectionAnalyzer*. Uma delas é a *StatementAnalyzer*. Ela, além de checar a validade da seleção, verifica se o conteúdo selecionado pode ser convertido em uma lista de objetos do tipo *Statement*, ou seja, aqueles que fazem parte do corpo de um método. De agora em diante, referenciaremos esses últimos objetos apenas como *statements*.

Criamos uma classe *ExtractRefactoringAnalyzer* (Figura 4.10), subclasse de *StatementAnalyzer*, que complementa esse *visitor* com mecanismos que facilitam sua utilização.

Antes de instanciar essa classe, criamos um objeto do tipo *Selection* a partir do objeto do tipo *ITextSelection* que obtivemos da etapa anterior.

```
ITextSelection text = ...;
Selection selection = Selection.createFromStartLength(text.getOffset(),
    text.getLength());
```

Com o objeto referenciado por “*selection*”, podemos executar a operação a seguir:

```
ExtractRefactoringAnalyzer eRAnalyzer =
    ExtractRefactoringAnalyzer.create(iCU, selection);
cU.accept(eRAnalyzer);
List statementList = eRAnalyzer.getStatementList();
```

Passamos nosso *visitor* para o método *accept* do objeto do tipo *CompilationUnit*, raiz da AST. Esse *visitor* fará verificações de validade da seleção e converterá os elementos do texto selecionado em uma lista de *statements* (Figura 4.11).

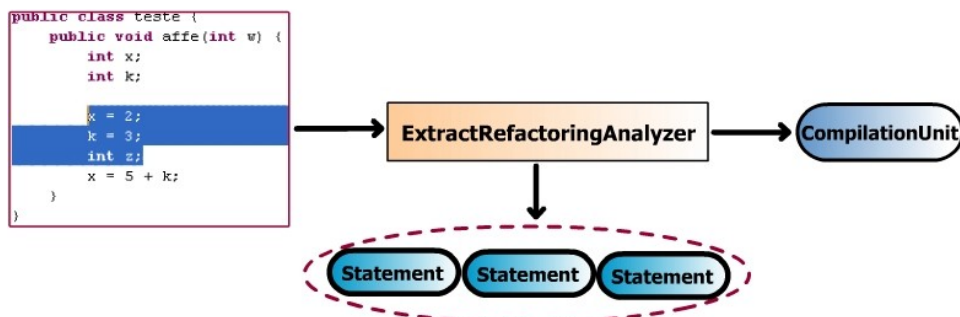


Figura 4.11: obtenção de uma lista de *statements*

O resultado da verificação da validade da seleção é armazenado em um objeto do tipo *RefactoringStatus*. Esse mecanismo já é implementado pelo *visitor* *StatementAnalyzer*. Apenas precisamos obter esse objeto do nosso *visitor* e verificarmos se ele registrou algum erro ocorrido durante o processo de análise da AST.

```
RefactoringStatus result = new RefactoringStatus();
result.merge(eRAnalyzer.getStatus());
```

Se o objeto de status não possuir erros e se a lista retornada não for vazia, podemos dar continuidade ao processo. Esse processo e outras verificações de pré-condições serão tratados em detalhes no capítulo de Implementação.

Encerrada essa etapa, a classe *SelectionActionRefactoring1* é instanciada pelo objeto do tipo *Refactoring1*. Ela tem a função de criar um novo aspecto ou atualizar um já existente. Isso depende da opção escolhida pelo usuário na etapa anterior. Ela também tem a função de obter dados que são importantes para a criação do código que será inserido dentro do aspecto, como listas de *imports* e exceções, método onde os *statements* selecionados estão, entre outros. Um diagrama de

classes simplificado dessa entidade e das demais relacionadas a ela é exibido na Figura 4.12.

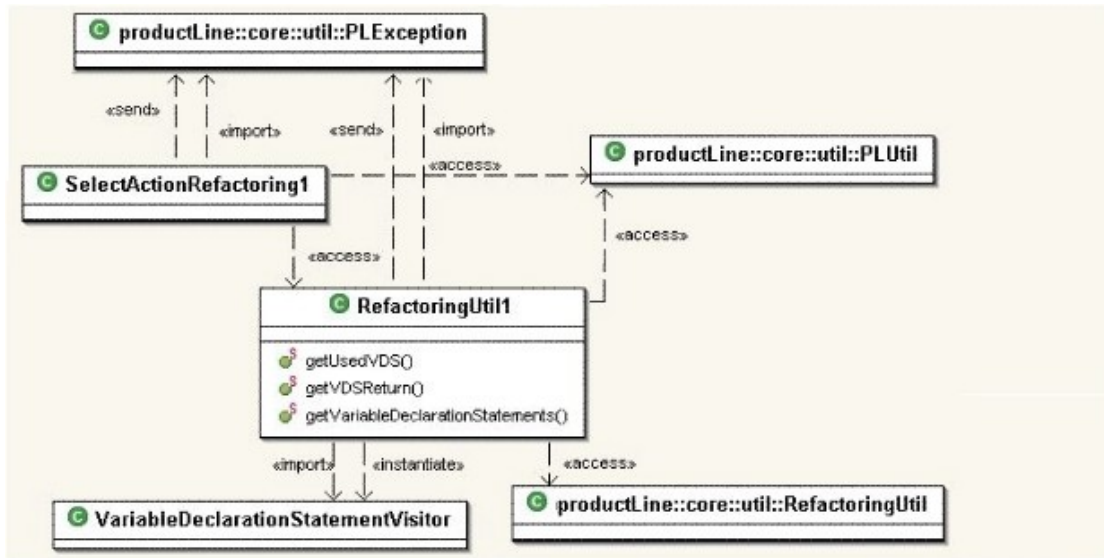


Figura 4.12: *SelectActionRefactoring1* e *RefactoringUtil1*

Baseado na referência ao arquivo do aspecto (obtida na primeira etapa, explanada em detalhes no próximo capítulo), a classe cria um novo aspecto (se a referência for nula) ou atualiza o aspecto (se realmente existir um arquivo referenciado). Essas últimas ações são executadas pela classe **RefactoringUtil1**, que também possui métodos que auxiliam na atualização da classe original após o processo de *refactoring*.

Os métodos que executam essas operações de criação do aspecto são o *generateAspect(...)* e o *updateAspect(...)*. Ambos chamam o método *generateAspectCode(...)*, que retornará o código que deverá ser inserido no aspecto existente, ou novo.

O código final é gerado como texto e inserido diretamente no *buffer* do arquivo do aspecto. Nesse processo, não chegamos a gerar uma AST para o aspecto, como fizemos com o arquivo *.java*, por ausência de ferramentas para isso. Utilizamos, no nosso desenvolvimento, algumas funcionalidades do *plug-in* AJDT, mencionado no capítulo 2. Esse *plug-in* ainda é muito restrito, não oferecendo o mesmo suporte para análise de AST's que o JDT possui. Até o encerramento deste trabalho, o *plug-in* estava na versão 1.2.0 para o Eclipse 3.0. Através dos desenvolvedores Matt Chapman, Adrian Colyer e Andrew Clement, da equipe do AJDT, que ajudaram bastante no desenvolvimento deste trabalho, descobrimos que essa funcionalidade está na fila dos próximos requisitos que serão implementados na ferramenta, e que já foi requisitada por diversos desenvolvedores. Com esse novo requisito, poderemos adaptar a nossa ferramenta de forma que ela fique mais robusta e confiável, pois operações diretamente na AST são bem mais seguras do que operações com texto, além de implementar outros *refactorings* que são dependentes dessa funcionalidade.

Finalmente, a classe *SelectionActionRefactoring* substitui o conjunto de *statements* selecionados por uma chamada de método ao *inter-type declaration* que foi criado no aspecto. Já que temos uma lista de referências aos *statements* da árvore, a substituição torna-se um processo trivial de apagar os nós já existentes e adicionar um novo nó (do tipo invocação de método) no lugar.

5. Implementação

Neste capítulo, iremos entrar em detalhes da implementação do *Refactoring Extract Method to Aspect*. Os demais *refactorings* terão seu processo explicado de uma maneira mais geral, apenas mostrando a idéia do algoritmo usado.

No total, a ferramenta possui 43 classes e aproximadamente 9000 linhas de código. A principal parte da implementação está nos pacotes ***productLine.actions***, ***productLine.core.util***, ***productLine.core.util.checkers*** e ***productLine.core.refactoringX***, onde X representa um número identificador de cada *refactoring*. Demais pacotes tratam de configurações da interface gráfica da ferramenta.

5.1 Implementação do *Refactoring Extract Method to Aspect*

Vamos analisar o código do *Refactoring Extract Method to Aspect*, complementando pontos que não foram tratados no capítulo de arquitetura.

Refactoring1ActionDelegate

A execução do *refactoring* começa na classe *Refactoring1ActionDelegate*, como vimos no capítulo anterior. Quando o usuário executa a ação de *refactoring*, o sistema delega a execução para essa classe. Ela consegue obter o conteúdo do arquivo .java aberto no editor e a seleção de texto desse arquivo feita pelo usuário, dados imprescindíveis para a operação. Depois disso, o método *run()* é chamado para dar início ao processo, executando os seguintes passos:

1. Chama um método da classe ***PLUtil*** (Figura 4.9) para obter uma lista dos aspectos existentes no mesmo pacote do objeto que implementa *ICompilationUnit*, obtido a partir do conteúdo do arquivo .java aberto no editor. Se não existir nenhum aspecto, a lista ficará vazia. Esse fator influencia na configuração da janela que será exibida para o usuário no passo 2;
2. Chama a janela de opções com a classe ***PLVPDialog***, que irá pedir as seguintes informações do usuário: qual o nome do aspecto que será criado e que receberá o resultado da operação e qual o nome do *inter-type declaration* (ponto de variação) que será criado no aspecto e chamado na classe no lugar do código selecionado. Se já existir algum aspecto dentro do pacote (detectado no passo anterior), será exibida uma segunda opção, permitindo ao usuário escolher um aspecto já existente, se desejar, para receber o resultado da operação;
3. Faz a checagem de pré-condições básicas, como: “um trecho de código precisar ser selecionado no arquivo .java” e “string vazia não é aceita como nome do ponto de variação ou o nome do aspecto”;
4. Se as pré-condições forem válidas, um método da classe ***PLUtil*** é chamado para obter um objeto que representa uma ligação para o arquivo do aspecto a partir do nome fornecido pelo usuário. Se o arquivo do aspecto já existir, o objeto ficará ligado a ele; caso contrário, o objeto ficará com uma ligação nula. Esse tipo de objeto é denominado de *handler*, cujo objetivo, nesse domínio, é dar suporte ao acesso ou criação de um arquivo. Esse fator influenciará a ação da classe *SelectionActionRefactoring1*, detalhada mais adiante;

5. Inicializa o objeto da classe **Refactoring1**, que irá executar a operação especificada na ação, passando para ele todas as informações que obtivemos até o momento: objeto que representa o arquivo .java, objeto do texto selecionado, nome do ponto de variação e o objeto que representa uma referência para o arquivo de aspecto que receberá o código resultante. Essa etapa será detalhada mais adiante;

```
Refactoring1 r1 = new Refactoring1(iCU, text, vPName, aspectFile);
r1.run();
```

6. Após a execução da operação, alguns *imports* podem não ser mais necessários. Um objeto da ferramenta JDT é instanciado e utilizado (*OrganizeImportsAction*) para remover os *imports* que não são mais necessários:

```
OrganizeImportsAction oIA = new
    OrganizeImportsAction(iEditorPart.getEditorSite());
oIA.run(this.iCU);
```

7. Por fim, ele abre no editor o arquivo correspondente ao aspecto:

```
IDE.openEditor(iWorkbenchPage, aspectFile, true);
```

Os erros que ocorrerem durante a execução são salvos no arquivo **ProductLineLog**.

Refactoring1

Depois de analisar o comportamento da classe associada diretamente à ação solicitada pelo usuário, resta compreender o funcionamento das classes que fazem o diferencial do nosso *plug-in*. Vamos continuar com a classe *Refactoring1*, chamada no passo 5 do fluxo anterior. Ela pode ser vista com suas classes relacionadas no diagrama da Figura 4.10.

Assim como a classe anterior, todo o comportamento principal dessa classe está no método *run()*. Esse padrão foi usado com frequência no desenvolvimento da ferramenta.

O primeiro passo é obter um objeto do tipo *CompilationUnit* do arquivo .java para podermos capturar a lista de *statements* selecionados pelo usuário. Esse processo já foi explicado previamente. Um *visitor* é responsável por obter a lista, sendo utilizado como um objeto da classe *ExtractRefactoringAnalyzer*, desenvolvida nesse projeto no pacote **productLine.core.util**, subclasse de *StatementAnalyzer*, mencionada no capítulo anterior. Essa classe simplesmente implementa métodos do tipo *endVisit* (capítulo 4.1.1) para todos os *statements* que devem ser capturados durante a navegação da AST. Enquanto isso, é feita a verificação da validade da seleção do código do usuário, processo implementado em *StatementAnalyzer* e usado por nossos métodos. Um exemplo de um deles pode ser visto a seguir, para o tipo *IfStatement*.

```
public void endVisit(IfStatement node) {
    if(contains(getSelectedNodes(), node)) {
        statementList.add(node);
    }
}
```

}

Nesse caso, o método *endVisit* irá analisar todos os objetos do tipo *IfStatement* do objeto do tipo *CompilationUnit*, que corresponde à raiz da AST do nosso arquivo *.java*. Com o método *getSelectedNodes()*, implementado na superclasse *SelectionAnalyzer*, obtemos a lista de todos os nós que foram selecionados (a partir de uma análise do objeto do tipo *Selection* recebido). Em seguida, verificamos se o nosso objeto do tipo *IfStatement* capturado pertence a essa seleção através do método *contains*. Se pertencer, adicionamos esse *statements* à nossa lista.

A verificação da validade da seleção é feita no método *endVisit(CompilationUnit)* redefinido em *StatementAnalyzer*. Essa verificação irá conferir se a seleção do usuário apenas possui *statements*. Na chamada do método *endVisit(IfStatement)*, essa verificação já tinha sido executada.

Se quiséssemos obter apenas objetos do tipo *IfStatements* da seleção, o único método da classe seria o exibido acima. Como esse não é o caso, redefinimos esse método para todas as classes que são subclasses da classe *Statement*. Não definimos um *endVisit* diretamente para *Statement* porque ela é uma classe abstrata, não tendo um método definido, conseqüentemente, na superclasse *ASTVisitor* (Figura 5.1).

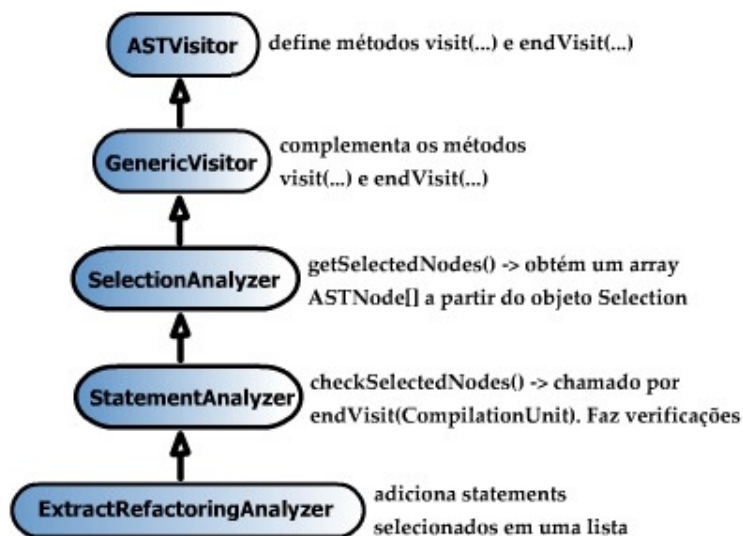


Figura 5.1: principais funções dos visitantes

Depois de verificar que a operação não possui erros, através do objeto de status retornado pelo *visitor*, e que a lista obtida não está vazia, ou seja, o usuário selecionou algo válido, entramos numa nova etapa de checagem de pré-condições mais específicas do *refactoring*.

Em contrapartida com os outros *refactorings*, que chamam a classe *Checker* do pacote ***productLine.core.util.checkers*** para avaliar pré-condições mais específicas da operação, essa classe aproveita a checagem de uma pré-condição já existente em um *refactoring* do JDT: o *Extract Method*.

Essa pré-condição é: o bloco de código selecionado não pode atualizar mais de uma variável local que é utilizada no bloco seguinte (mais detalhes podem ser vistos no capítulo 3). Essa operação pode ser vista a seguir:

```
ExtractMethodRefactoring eMRefactoring =
    ExtractMethodRefactoring.create(iCU, selection.getOffset(),
        selection.getLength(),
        JavaPreferencesSettings.getCodeGenerationSettings());
result.merge(eMRefactoring.checkInitialConditions(null));
```

Utilizamos o método *checkInitialConditions*, da classe *ExtractMethodRefactoring* do *refactoring Extract Method*. Esse método faz a checagem dessa pré-condição, poupando-nos trabalho de ter que implementar essa verificação. O método retorna um objeto do tipo *RefactoringStatus*, que aproveitamos para adicionar seu status ao nosso objeto de status referenciado pela variável *result*. Mais detalhes desse objeto de status podem ser vistos no capítulo anterior.

Se o objeto de status não possuir nenhum aviso de erro, damos andamento à execução chamando a classe *SelectActionRefactoring1*.

SelectActionRefactoring1

Essa classe tem a função de criar um novo aspecto ou atualizar um já existente. Isso depende da opção escolhida pelo usuário na etapa inicial. O método *run()* dessa classe segue o seguinte fluxo de eventos:

1. Chama o método *getVariableDeclarationListAndReturn()*, que tem a função de obter a possível lista de declaração de variáveis do ponto de variação. Isso é necessário quando o bloco selecionado acessa variáveis locais, pois elas precisam ser passadas do método da classe para a declaração do ponto de variação que está no aspecto (como um *inter-type declaration*). Esse método também obtém a variável de retorno, se for necessário retornar algum valor de variável local ao final da execução;
2. Obtém o método que possui o código selecionado, através de um método da classe **PLUtil** (Figura 4.12);

```
MethodDeclaration parentMD = (MethodDeclaration)
    PLUtil.getParentMD(stTemp);
```

3. Obtém lista dos possíveis *imports* que são usados na seleção, através de um método da classe **PLUtil**. Essa lista deve ser declarada no aspecto, se existirem classes de diferentes pacotes na nossa seleção;

```
importsList = PLUtil.getImportListFromStList(statementList, cU);
```

4. Obtém lista das possíveis exceções. Se nosso bloco lança algum conjunto de exceções, devemos definir o mesmo no *inter-type declaration* que irá conter esse bloco. Esse procedimento é executado também pela classe **PLUtil**;

```
List exceptionListS = PLUtil.getExceptionListFromStList(statementList,
    iCU);
```

5. Por fim, chamamos o método `selectAction()`, que definirá se a operação que será executada é de atualização de um aspecto ou criação de um novo. Esse método, inicialmente, verifica se a classe do código selecionado já possui alguma chamada a um ponto de variação com o mesmo nome que foi passado pelo usuário. Não existirem chamadas como essa é uma pré-condição para a execução da operação. Em seguida, o histórico do aspecto deve ser apagado, pois isso, por uma maneira ainda desconhecida, bloqueia a ação do nosso *refactoring*.

```
aspectFile.clearHistory(null);
```

Em seguida, é feita a verificação se nossa variável de aspecto referencia algum arquivo de aspecto. Se referenciar, o aspecto será atualizado. Senão, criaremos um novo aspecto;

```
if(aspectFile.exists()) {
    updateAspectFile();
} else {
    createAspectFile();
}
```

6. Esses métodos serão tratados mais adiante. Encerramos nosso fluxo de execução com a operação de atualização da nossa classe. Ao invés do antigo código selecionado, teremos uma chamada ao ponto de variação do aspecto. Esse ponto de variação pode receber argumentos ou retornar um valor. Esses dados podem ser coletados do objeto do tipo *ExtractMethodRefactoring*, reaproveitado por nossa ferramenta, e da variável local que deve ser retornada, referenciada pela variável `vDSReturn`, obtida no passo 1. Se não for necessário retorno, o valor de `vDSReturn` é nulo.

```
RefactoringUtil.runOriginRefactoring(cU, statementList,
    variationPointName, eMRefactoring, vDSReturn);
```

Resta analisar os métodos `updateAspectFile` e `createAspectFile`. O primeiro, inicia sua execução obtendo o texto do aspecto existente através do método `getAspectString` de *PLUtil*.

```
String stOrigin = PLUtil.getAspectString(aspectFile);
```

Se o conteúdo desse texto não for vazio, chamamos o método `updateAspect` da classe *RefactoringUtil1*, responsável por gerar o código do *refactoring*, passando todos os argumentos que são necessários para esse procedimento, inclusive o texto obtido do aspecto que será atualizado.

```
String newStOrigin = RefactoringUtil1.updateAspect(codeSelected, stOrigin,
    importsList, exceptionListS, parentMD, variationPointName,
    eMRefactoring, vDSList, vDSReturn);
```

Esse método retorna um novo texto, que representa o aspecto existente com o código gerado. Essa operação é feita diretamente no texto, como explicamos no capítulo anterior, ao invés de criarmos uma AST para o aspecto e aplicarmos as

mudanças diretamente nela. Por fim, passamos o novo texto para o aspecto da seguinte forma:

```
PLUtil.updateAspect(aspectFile, newStOrigin);
```

Utilizando o método *updateAspect* de **PLUtil**, substituímos o texto antigo do arquivo de aspecto pelo novo. O segundo método, *createAspectFile*, gera primeiro o código resultante do aspecto.

```
String s0 = RefactoringUtil1.generateAspect(this.codeSelected, aspectName,
    importsList, exceptionListS, parentMD, variationPointName,
    emRefactoring, vDSLList, vDSReturn);
```

Em seguida, esse texto é passado para o método *createNewAspect* de **PLUtil**, que cria o arquivo de aspecto e insere o texto dentro dele:

```
aspectFile = PLUtil.createNewAspect(iCU, aspectName, s0);
```

Quando as funcionalidades de manipulação de AST's de aspectos forem implementadas no AJDT, precisamos apenas mudar os métodos *updateAspectFile* e *createAspectFile*, além dos métodos da classe *RefactoringUtil1* que estão associados a eles, de forma a tornar a ferramenta mais robusta.

A alteração que será aplicada à classe de origem ainda está no objeto do tipo *CompilationUnit*. Para refletir essa alteração no objeto que implementa a interface *ICompilationUnit*, chamamos o método de escrita de **PLUtil** que escreve o conteúdo final no arquivo .java.

```
PLUtil.writeCompilationUnit(cU, iCU);
```

RefactoringUtil1

Resta entrarmos nos detalhes na classe que gera o código dos aspectos e atualiza a classe de origem. A atualização da classe de origem é relativamente simples, sendo efetuada apenas com algumas manipulações da AST.

O método *generateAspect* e o *updateAspect* chamam o método *generateAspectCode* para gerar o código que deverá ser inserido no aspecto. A maior diferença entre eles dois é que o primeiro cria o corpo do aspecto.

```
String aspectCode = "privileged aspect " + aspectName + " {\n" + text +
    "\n}";
aspectCode = packageName + imports + aspectCode;
```

Onde “*text*” corresponde ao texto gerado pelo método *generateAspectCode*. Os textos correspondentes ao nome do pacote e à lista de imports são obtidos por métodos auxiliares da classe **RefactoringUtil** (Figura 4.12). O segundo, pelo contrário, atualiza o aspecto já existente com:

```
String aspectCode = RefactoringUtil.updateAspectString(source, text);
```

Onde “source” corresponde ao código da origem e “text” corresponde ao texto gerado pelo método *generateAspectCode*. O fluxo do método *generateAspectCode* é exibido a seguir:

1. É gerada a assinatura do método. Esse procedimento é delegado para a classe *RefactoringUtil*.

```
String signature = RefactoringUtil.generateSignature(parentMD,  
    eMRefactoring);
```

O primeiro argumento corresponde ao método que contém a seleção. O segundo, ao objeto do tipo *ExtractMethodRefactoring*, do JDT, de onde extraímos algumas informações para a geração da assinatura;

2. Do objeto do tipo *ExtractMethodRefactoring* também conseguimos obter a lista de parâmetros. Se a lista de parâmetros for vazia, essa lista será vazia e nada será impresso no código final referente a parâmetros.

```
List paramList = eMRefactoring.getParameterInfos();
```

3. Convertemos para texto nossa lista de exceções.

```
String exceptionSt = RefactoringUtil.printExceptionList(exceptionListS);
```

4. Criamos nosso texto que equivale à linha de retorno, se precisarmos retornar algum valor de variável local;
5. Por fim, montamos o código.

```
String aspectCode = "    " + signature + exceptionSt + "{\n        " +  
    declarations + codeSelected + returnString + "\n    }";
```

O objeto do tipo *String* referenciado pela variável “*aspectCode*” será retornado pelo método para ser colocado dentro do texto do aspecto por *generateAspect* ou *updateAspect*.

5.2 Implementação dos outros *refactorings*

A seguir, apresentamos um esquema simplificado da implementação dos demais *refactorings*. Pelo fato da estratégia de implementação dos *refactorings* ser bastante semelhante, iremos discorrer sobre o fluxo dessas outras funcionalidades de uma maneira mais categórica do que a primeira.

5.2.1 Refactoring Extract Resource to Aspect - after

Esse *refactoring* extrai uma atribuição de um atributo da classe para o aspecto, juntamente com sua declaração. No aspecto, a declaração do atributo vira *inter-type declaration* e a atribuição é inserida dentro de um *advice*. Esse procedimento pode ser aplicado também a uma lista de atribuições consecutivas. Mais detalhes sobre seu funcionamento podem ser vistos no capítulo 3.

Os passos principais da sua implementação serão tratados a seguir.

1. O *refactoring* inicia na classe *Refactoring2ActionDelegate*. Da mesma forma que a classe *Refactoring1ActionDelegate*, essa classe inicia o processo após a solicitação da operação pelo usuário, capturando o arquivo .java que está aberto no editor juntamente com o código selecionado nele. Nessa etapa, uma janela é aberta, perguntando ao usuário se ele quer gerar o código correspondente ao *refactoring* em um aspecto já existente ou se quer gerar em um novo aspecto. Esse processo diferencia-se apenas do *Refactoring1ActionDelegate* por não solicitar um nome para o ponto de variação, já que essa estrutura concerne apenas ao *refactoring Extract Method to Aspect*. O restante do processo é executado pelo método *run()*, que tem comportamento bastante similar ao método *run()* de *Refactoring1ActionDelegate*, tendo como única diferença que a classe *Refactoring2* é usada, ao invés da *Refactoring1*;
2. O comportamento de *Refactoring2* é definido pelo método *run()*. Ele cria, inicialmente, um objeto do tipo *CompilationUnit* a partir do objeto que implementa a interface *ICompilationUnit*, obtido da etapa anterior. Esse objeto do tipo *CompilationUnit*, juntamente com o objeto que representa a seleção de texto do usuário, é útil para capturarmos a seleção na forma de uma lista estruturada, ao invés de texto. Esse processo procede da mesma maneira do *refactoring Extract Method to Aspect*, desde o momento em que o *visitor ExtractRefactoringAnalyzer* é utilizado até a verificação se a seleção foi válida através do status e da lista retornados por ele;
3. A lista obtida do *visitor* é analisada. Apenas atribuições a atributos da classe são permitidas nessa lista. Qualquer estrutura diferente invalida a operação. Através do *visitor AssignmentFieldHandle*, específico para esse caso, é possível analisar essa primeira pré-condição, além de obter uma estrutura que representa a declaração do atributo da classe que está sendo atualizada (*VariableDeclarationFragment*). Adicionamos essa estrutura em uma lista, juntamente com o *statement* que corresponde à atribuição. Essa nova lista armazena todos os pares desse tipo que forem encontrados nesse processo de análise da primeira lista (Figura 5.1). Ela será passada para o objeto da classe *SelectActionRefactoring2*, que será explanado a seguir;

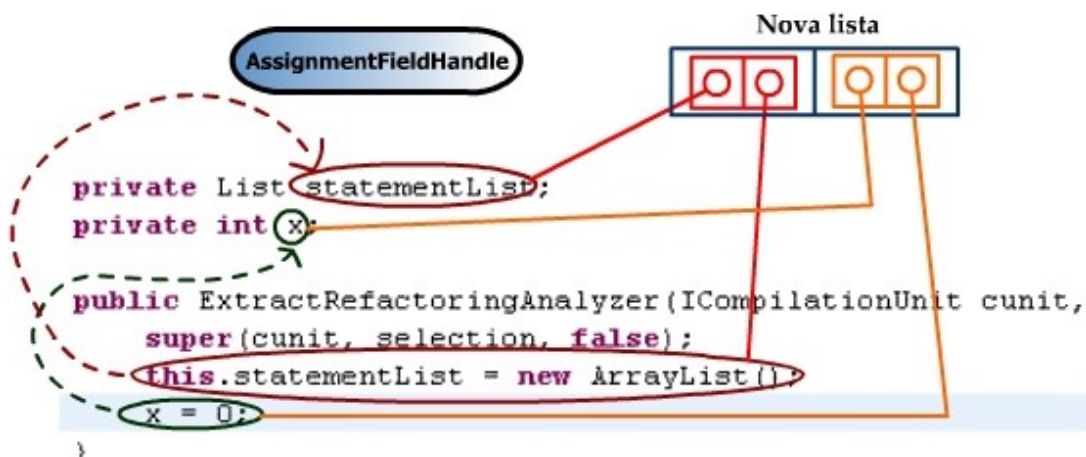


Figura 5.1: *AssignmentFieldHandle*

4. Depois de verificarmos o objeto de status, retornado pelo último *visitor*, passamos para a etapa de obtenção do método que contém o código selecionado. Esse objeto é útil para a checagem de pré-condições mais específicas dessa operação. Ele é passado como argumento para o método *check2*, da classe *Checker* do pacote ***productLine.core.util.checkers***, específico para analisar pré-condições de domínio mais específico. Esse método irá verificar, em primeiro lugar, se o bloco selecionado não utiliza variáveis locais ou chama construções do tipo *super*. Em seguida, ele verifica se os atributos que estão sendo atualizados no bloco não são utilizados em nenhum outro lugar da classe fora daquela seleção. O método *check2* retorna um objeto de status. Se a seleção atender a todas as pré-condições (objeto de status não possui nenhuma mensagem de erro), o *refactoring* pode ser executado. O próximo passo é criar um objeto do tipo *SelectActionRefactoring2*;
5. O objeto *SelectActionRefactoring2*, da mesma forma que o *SelectActionRefactoring1*, tem a função de criar um novo aspecto ou atualizar um já existente. Isso depende da opção escolhida pelo usuário na etapa inicial. Além de obter possíveis listas de *imports* e exceções, essa classe usa um *visitor* de nome *UpdateAttributeCthisVisitor* para inserir em todas as utilizações de atributos da classe, que foram movidos para o aspecto, o identificador *cthis*. Esse identificador representará a classe. Isso é essencial porque esses atributos só são acessíveis dentro do aspecto com um identificador como esse. Por fim, o método *updateAspectFile* ou *createAspectFile* é chamado para gerar o aspecto. Essa operação de geração de código é relativamente simples de entender, sendo apenas um processo de manipular objetos do tipo *String* para obter um texto final que representa o código que será inserido no aspecto;
6. Demais processos de escrita no arquivo de aspecto e atualização da classe ocorrem da mesma maneira explanada no *refactoring Extract Method to Aspect*. A atualização da classe de origem é dada no objeto da classe *Refactoring2*, ao final dessa operação. Essa atualização envolve apenas a remoção dos objetos de atribuição e declaração dos atributos envolvidos no processo.

5.2.2 Refactoring Extract Context

Esse *refactoring* extrai o contexto do código selecionado pelo usuário. Esse contexto é composto por estruturas de controle aninhadas. Mais detalhes sobre seu funcionamento podem ser vistos no capítulo 3.

Os primeiros passos deste *refactoring* são semelhantes aos passos 1 e 2 do *refactoring Extract Resource to Aspect – after*, explicado anteriormente. A única diferença é a utilização da classe *Refactoring3*, ao invés da *Refactoring2*. Os demais passos da sua implementação serão tratados a seguir.

1. Depois de obter a lista de seleção, o método que contém o código selecionado é obtido. Com ele, podemos dar início à verificação de pré-condições com o método *check3* da classe *Check* do pacote ***productLine.core.util.checkers***. A primeira pré-condição verificada é que, além do bloco selecionado e seu contexto, o método não pode ter mais nenhuma outra estrutura. A segunda é que o bloco selecionado não declara ou usa variáveis locais do contexto e nem

- chama construções do tipo *super*. Essa segunda operação é reusada pelo *refactoring* anterior. Se a seleção atender a todas as pré-condições (objeto de status não possui nenhuma mensagem de erro), o *refactoring* pode ser executado. O próximo passo é criar um objeto do tipo *SelectActionRefactoring3*;
2. Da mesma forma que a classe *SelectActionRefactoring2*, essa classe coleta dados importantes para o processo de *refactoring* e chamada os métodos da classe *RefactoringUtil3* para criar ou atualizar um aspecto;
 3. Ao final dessas operações de criação dos aspectos, a classe de origem é alterada em *Refactoring3*. Apenas o contexto é removido, ou seja, as estruturas de controle que envolvem o código selecionado.

A operação de obtenção de um contexto é relativamente simples. Vamos discorrer brevemente sobre esse processo por essa etapa ser importante para esse caso.

Com qualquer *statement* da minha lista de *statements* eu consigo obter as estruturas de controle que envolvem essa lista. Qualquer estrutura de controle possui um objeto *Block*, com uma lista de *statements*, assim como uma declaração de método (Figura 2.6). A partir de qualquer nó da minha AST, posso obter uma referência para o seu nó “pai” através do método *getParent()*, ou seja, aquele nó que está acima da hierarquia. Utilizando esse método recursivamente com um laço *while*, eu posso capturar a estrutura de controle mais externa à minha seleção. É só interromper a execução do *while* quando eu encontrar um nó do tipo *MethoDeclaration*, ou seja, o método que contém minha seleção. Um trecho de código dessa execução pode ser visto a seguir:

```
ASTNode astNode = statement;
while(astNode.getParent().getParent().getNodeTypes()
    != ASTNode.METHOD_DECLARATION) {
    astNode = astNode.getParent();
}
```

Um *statement* da minha lista é referenciado pela variável de mesmo nome. Através do seu método *getParent()*, eu consigo obter seu *Block*. Com uma chamada novamente a *getParent()* no *Block*, eu consigo obter a estrutura de controle ou a declaração do método que contém minha lista. Executando a verificação do “pai” do *Block* dentro de um laço *while*, eu consigo pegar a estrutura de controle mais externa da minha seleção, aquela que é um *statement* da declaração do método.

5.2.3 Refactoring Extract Before Block

Extrai o primeiro bloco de código de um método ou construtor. Mais detalhes sobre seu funcionamento podem ser vistos no capítulo 3.

Os primeiros passos deste *refactoring* são semelhantes aos passos 1 e 2 do *refactoring Extract Resource to Aspect – after*, explicado anteriormente. Os demais passos da sua implementação serão tratados a seguir.

1. De posse da lista que contém os *statements* selecionados, acesso o método que a contém e depois pego a lista de todos os *statements* desse método. Verifico, então, se minha lista faz parte do começo da lista de *statements* do método. Se

- fizer, posso dar continuidade ao processo, pois isso garante que minha seleção de código está no começo do método;
2. Esse passo é semelhante ao passo 1 do *refactoring Extract Context*. Com a diferença de que o método chamado na classe *Checker* é o *check45*, que serve também para verificar pré-condições do *refactoring Extract After Block*. Esse método apenas verifica se o código selecionado não declara e não usa variáveis locais, além de não chamar construções do tipo *super*. Se a seleção atender a todas as pré-condições (objeto de status não possui nenhuma mensagem de erro), o *refactoring* pode ser executado. O próximo passo é criar um objeto do tipo *SelectActionRefactoring4*;
 3. Da mesma forma que a classe *SelectActionRefactoring2*, essa classe coleta dados importantes para o processo de *refactoring* e chama os métodos da classe *RefactoringUtil4* para criar ou atualizar um aspecto. Esse passo é semelhante ao passo 5 do *refactoring Extract Resource to Aspect – after*, chegando a utilizar também classes como *UpdateAttributeCthisVisitor*;
 4. Depois de criar o código do aspecto, atualizo minha classe excluindo todo o bloco selecionado. Por fim, escrevo as alterações do objeto do tipo *CompilationUnit* no objeto que implementa *ICompilationUnit*, que representa meu arquivo .java.

5.2.4 Refactoring Extract After Block

Extraí o último bloco de código de um método ou construtor. Mais detalhes sobre seu funcionamento podem ser vistos no capítulo 3. O funcionamento desse *refactoring* é extremamente semelhante ao *Refactoring Extract Before Block*. As diferenças estão em algumas pré-condições verificadas. Dentre elas, a mais notável é a que o trecho de código que será extraído deve ser o último bloco de código do seu método, em contrapartida com o caso anterior, que extraí apenas o primeiro bloco.

5.2.5 Refactoring Change Class Hierarchy

Esse *refactoring* altera a superclasse de uma classe para a próxima superclasse da hierarquia. Mais detalhes sobre seu funcionamento pode ser visto no capítulo 3. Os passos principais da sua implementação serão explanados a seguir.

1. O *refactoring* inicia na classe *Refactoring6ActionDelegate*. Da mesma forma que a classe *Refactoring1ActionDelegate*, essa classe inicia o processo após a solicitação da operação pelo usuário, capturando o arquivo .java que está aberto no editor, mas sem o código selecionado nele. Nessa etapa, uma janela é aberta, perguntando ao usuário se ele quer gerar o código correspondente ao *refactoring* em um aspecto já existente ou se quer gerar em um novo aspecto. Esse processo diferencia-se dos demais por não envolver seleção de código. O restante do processo é executado pelo método *run()*, que tem comportamento bastante similar ao método *run()* de *Refactoring1ActionDelegate*, tendo como única diferença que a classe *Refactoring6* é usada, ao invés da *Refactoring1*;
2. O método *run()* da classe *Refactoring6* apenas cria um objeto da classe *SelectActionRefactoring7*. Depois disso, atualiza a classe do nosso arquivo .java, pegando sua declaração e alterando o nome da sua superclasse para a superclasse dessa superclasse;

3. Esse método cria um objeto do tipo *CompilationUnit* a partir do objeto que implementa *ICompilationUnit*. Desse objeto, ele pega sua lista de *types()* à procura do primeiro nó que for do tipo *TypeDeclaration*, que equivale à declaração do método (Figura 2.5). De posse desse nó, obtemos através de um método dessa classe quem é sua superclasse. Também obtemos a lista de *imports* que serão necessários. De posse desses dois dados, podemos chamar os métodos *updateAspect* ou *createAspect*, dependendo da opção do usuário. Daí em diante, a geração do código do aspecto ocorre da mesma maneira explanada nos *refactorings* anteriores.

6. Cenários de uso

Neste capítulo, apresentamos os cenários de uso da nossa ferramenta. De uma forma mais categórica, serão explicados todos os passos para executar cada *refactoring* com sucesso.

6.1. Refactoring Extract Method to Aspect

O *refactoring Extract Method to Aspect* já foi explicado no capítulo 4. Os demais *refactorings* serão explicados seguindo a mesma sistemática dele.

Cada botão da ferramenta possui um *tooltip* associado que diz qual *refactoring* está associado a sua ação. Esses botões também estão disponíveis na Barra de Menu, através da opção ProductLine.

6.2. Refactoring Extract Resource to Aspect - after

1. Selecione um trecho de código de um arquivo Java™ aberto em um editor e clique no botão *Refactoring Extract Resource to Aspect - after*. Esse trecho selecionado deve fazer parte do corpo de um método ou de um construtor e que, além disso, dentro dele só existam atribuições a atributos da classe (Figura 6.1);

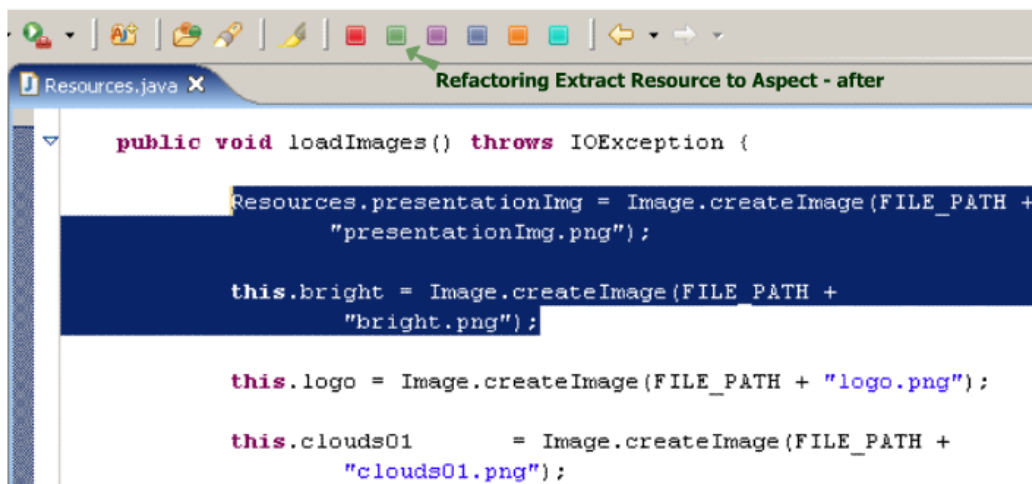


Figura 6.1: seleção de trecho de código

2. Uma caixa de diálogo é aberta, perguntando se você quer gerar o código correspondente ao *refactoring* em um aspecto já existente ou se quer gerar em um novo aspecto. Como não há nenhum aspecto criado no projeto, apenas a segunda opção está disponível (Figura 6.2);

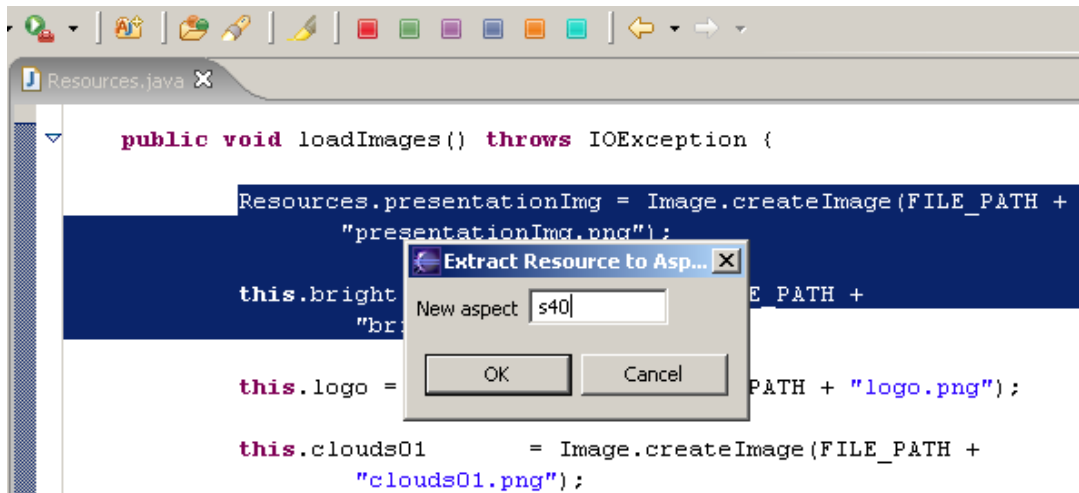


Figura 6.2: seleção de opção

3. A ferramenta executa a verificação das pré-condições no código selecionado com o objetivo de garantir que o código gerado vai estar sintaticamente e semanticamente correto. As atribuições e seus atributos são removidos da classe (Figura 6.3);

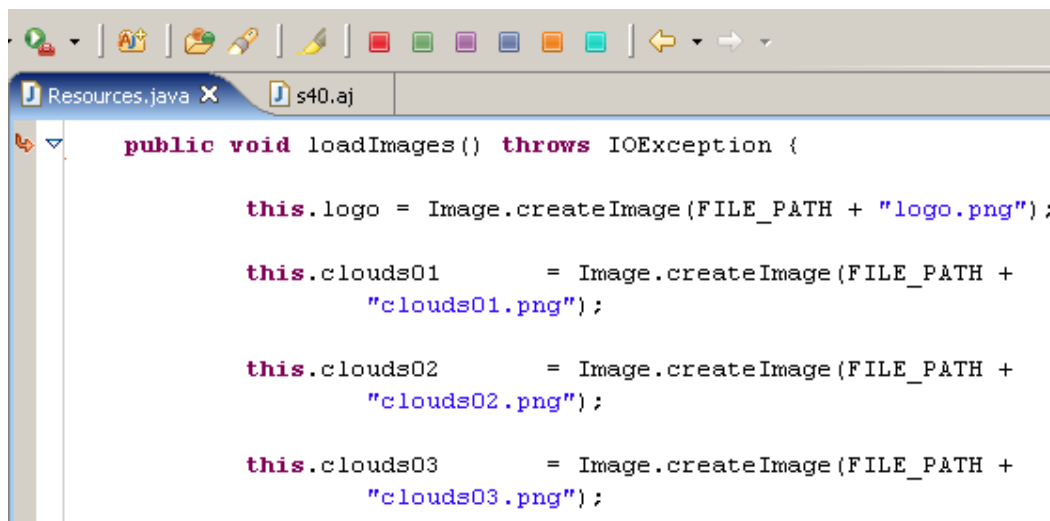
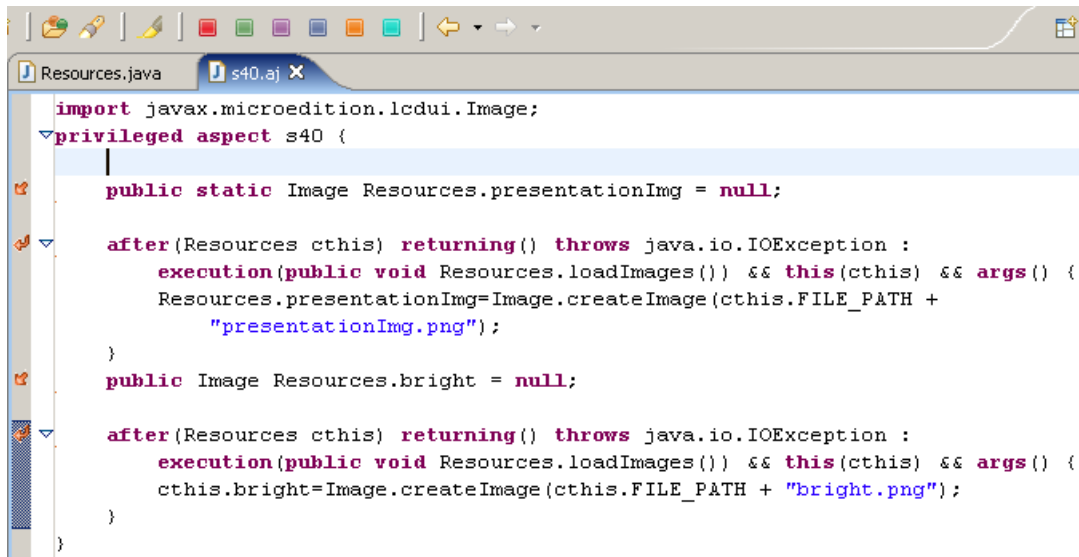


Figura 6.3: arquivo .java alterado

4. O código em *AspectJ* é gerado e escrito em um aspecto já existente ou em um novo aspecto, dentro do mesmo pacote do arquivo .java, conforme a opção escolhida no passo 2 (Figura 6.4).



```
import javax.microedition.lcdui.Image;
privileged aspect s40 {
    public static Image Resources.presentationImg = null;

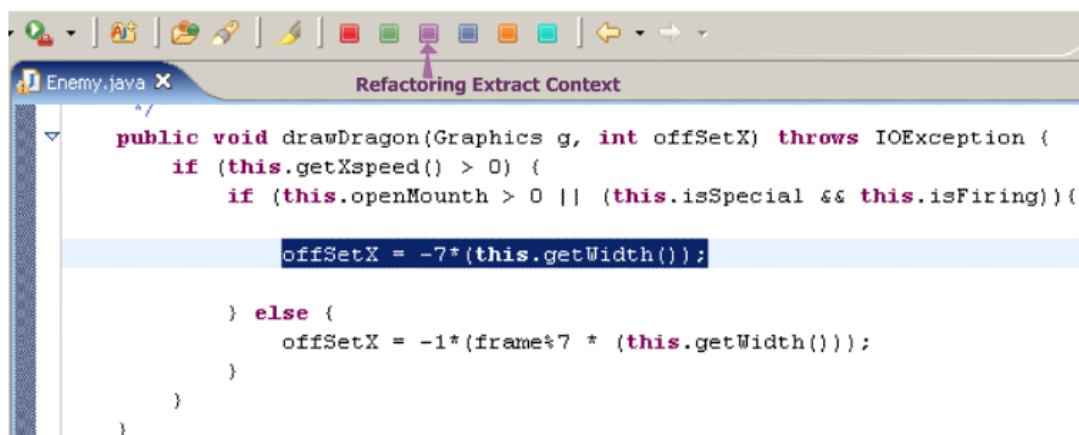
    after(Resources cthis) returning() throws java.io.IOException :
        execution(public void Resources.loadImages() && this(cthis) && args() {
        Resources.presentationImg=Image.createImage(cthis.FILE_PATH +
            "presentationImg.png");
        }
    public Image Resources.bright = null;

    after(Resources cthis) returning() throws java.io.IOException :
        execution(public void Resources.loadImages() && this(cthis) && args() {
        cthis.bright=Image.createImage(cthis.FILE_PATH + "bright.png");
        }
    }
}
```

Figura 6.4: código gerado

6.3. Refactoring Extract Context

1. Selecione um trecho de código de um arquivo Java™ aberto em um editor e clique no botão *Refactoring Extract Context*. Esse trecho selecionado deve fazer parte do corpo de um método ou de um construtor e, além disso, ele deve estar inserido, e somente ele, dentro do corpo de alguma estrutura de controle (ex. laços *if* e *while* e bloco *try-catch*). Essa estrutura de controle pode estar inserida também em outra estrutura de controle. Nenhuma outra estrutura pertencente ao método pode estar fora desse bloco de estruturas de controle aninhadas. Vale salientar que o código que está inserido na cláusula *else* também será extraído juntamente com sua estrutura de controle (Figura 6.5);



```
public void drawDragon(Graphics g, int offSetX) throws IOException {
    if (this.getXspeed() > 0) {
        if (this.openMounth > 0 || (this.isSpecial && this.isFiring)){
            offSetX = -7*(this.getWidth());
        } else {
            offSetX = -1*(frame%7 * (this.getWidth()));
        }
    }
}
```

Figura 6.5: seleção de trecho de código

2. Uma caixa de diálogo é aberta, perguntando se você quer gerar o código correspondente ao *refactoring* em um aspecto já existente ou se quer gerar em um novo aspecto. Estamos assumindo que já existe um aspecto dentro do mesmo pacote dessa classe (Figura 6.6);

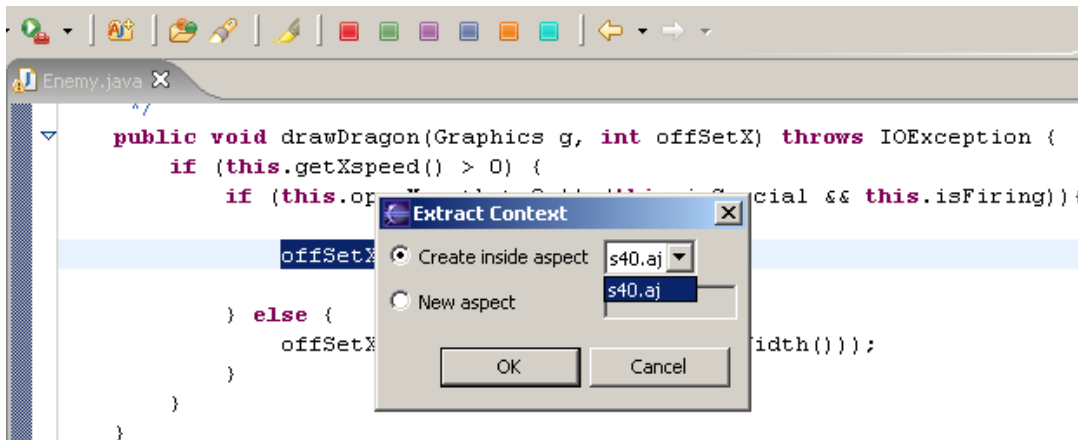


Figura 6.6: seleção de opção

3. A ferramenta executa a verificação das pré-condições no código selecionado com o objetivo de garantir que o código gerado vai estar sintaticamente e semanticamente correto. Todas as estruturas de controle que envolvem o bloco selecionado serão extraídas para o aspecto (Figura 6.7);

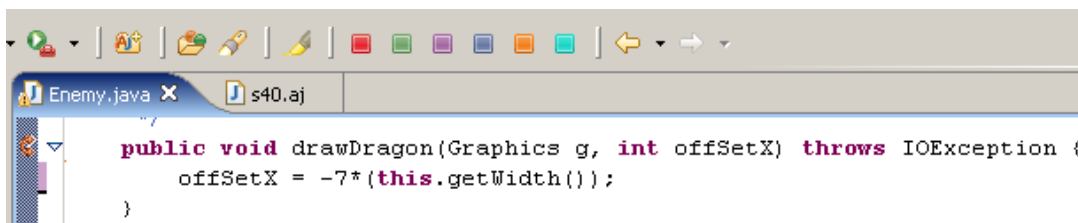


Figura 6.7: arquivo .java alterado

4. O código em *AspectJ* é gerado e escrito em um aspecto já existente ou em um novo aspecto, dentro do mesmo pacote do arquivo .java, conforme a opção escolhida no passo 2 (Figura 6.8).

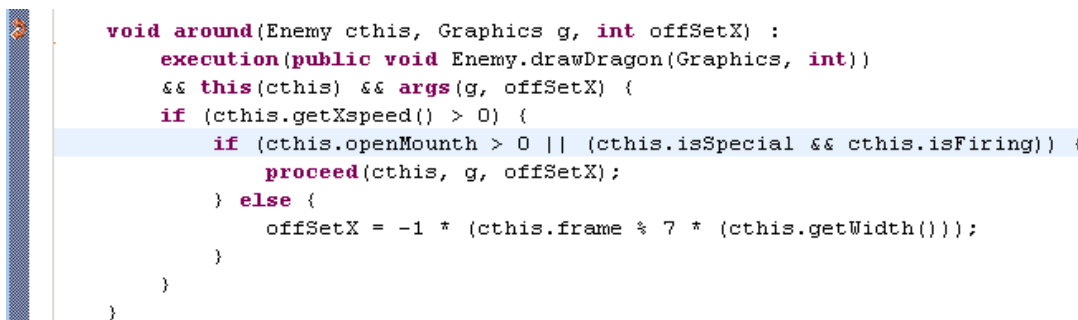


Figura 6.8: código gerado

6.4. Refactoring Extract Before Block

1. Selecione um trecho de código de um arquivo Java™ aberto em um editor e clique no em *Refactoring Extract Before Block*. O trecho deve corresponder ao primeiro bloco de código do método ou construtor (Figura 6.9);

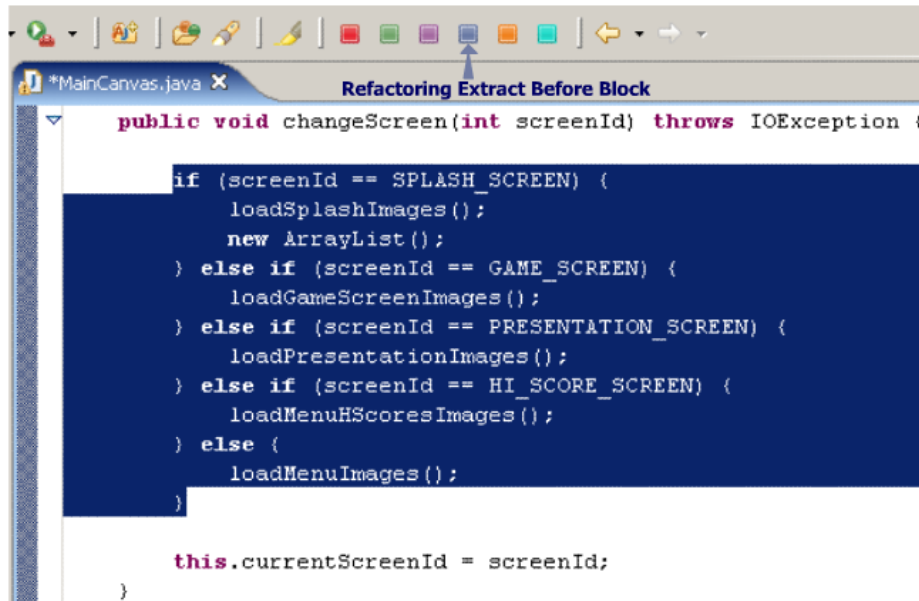


Figura 6.9: seleção de trecho de código

2. Uma caixa de diálogo é aberta, perguntando se você quer gerar o código correspondente ao *refactoring* em um aspecto já existente ou se quer gerar em um novo aspecto (Figura 6.10);

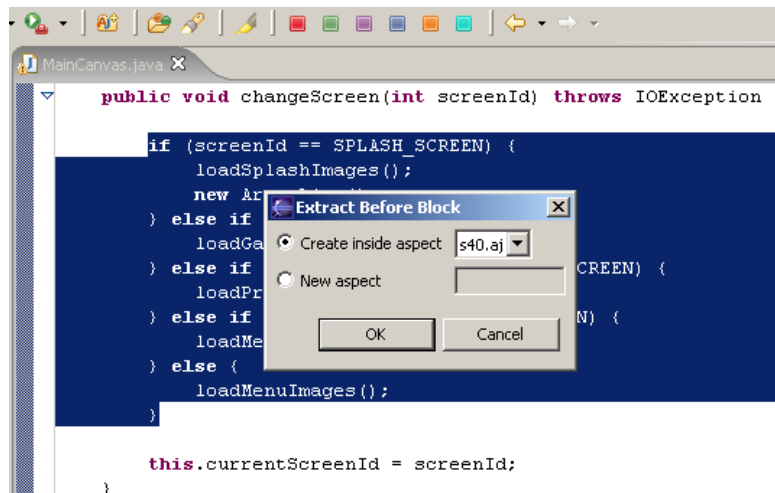


Figura 6.10: seleção de opção

3. A ferramenta executa a verificação das pré-condições no código selecionado com o objetivo de garantir que o código gerado vai estar sintaticamente e semanticamente correto. Esse código é removido do método (Figura 6.11);

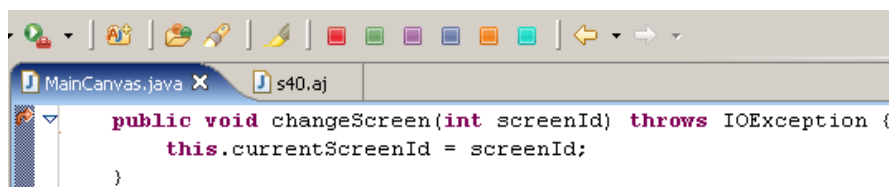


Figura 6.11: arquivo .java alterado

4. O código em *AspectJ* é gerado e escrito em um aspecto já existente ou em um novo aspecto, dentro do mesmo pacote do arquivo .java, conforme a opção escolhida no passo 2 (Figura 6.12).

```
before(MainCanvas cthis, int screenId) :
    execution(public void MainCanvas.changeScreen(int))
    && this(cthis) && args(screenId) {
    if (screenId == cthis.SPLASH_SCREEN) {
        cthis.loadSplashImages();
        new ArrayList();
    } else if (screenId == cthis.GAME_SCREEN) {
        cthis.loadGameScreenImages();
    } else if (screenId == cthis.PRESENTATION_SCREEN) {
        cthis.loadPresentationImages();
    } else if (screenId == cthis.HI_SCORE_SCREEN) {
        cthis.loadMenuHScoresImages();
    } else {
        cthis.loadMenuImages();
    }
}
```

Figura 6.12: código gerado

6.5. Refactoring Extract After Block

1. Selecione um trecho de código de um arquivo Java™ aberto em um editor e clique em *Refactoring Extract After Block*. O trecho deve corresponder ao último bloco de código executado pelo método ou construtor (Figura 6.13);

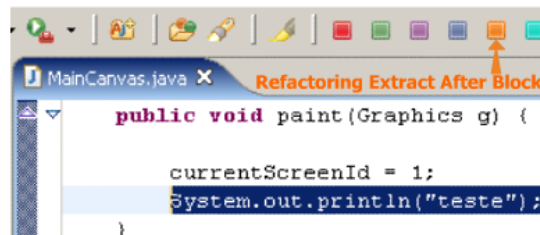


Figura 6.13: seleção de trecho de código

2. Uma caixa de diálogo é aberta, perguntando se você quer gerar o código correspondente ao *refactoring* em um aspecto já existente ou se quer gerar em um novo aspecto (Figura 6.14);

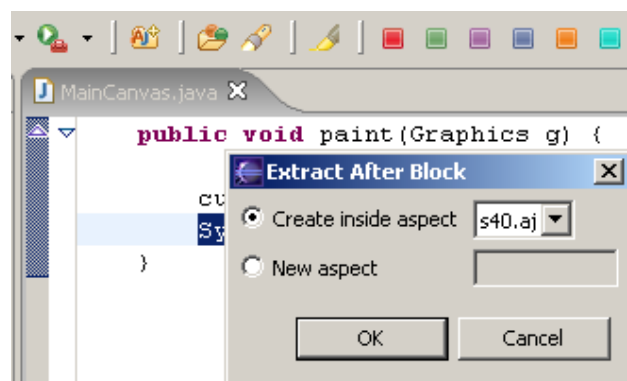


Figura 6.14: seleção de opção

3. A ferramenta executa a verificação das pré-condições no código selecionado com o objetivo de garantir que o código gerado vai estar sintaticamente e semanticamente correto. Esse código é removido da classe (Figura 6.15);

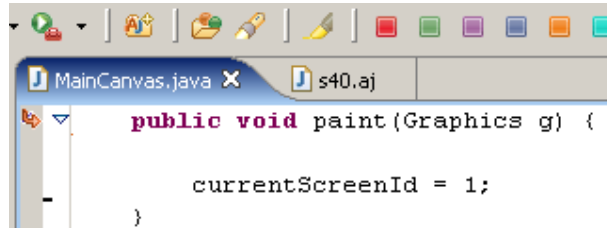


Figura 6.15: arquivo .java alterado

4. O código em *AspectJ* é gerado e escrito em um aspecto já existente ou em um novo aspecto, dentro do mesmo pacote do arquivo .java, conforme a opção escolhida no passo 2 (Figura 6.16).

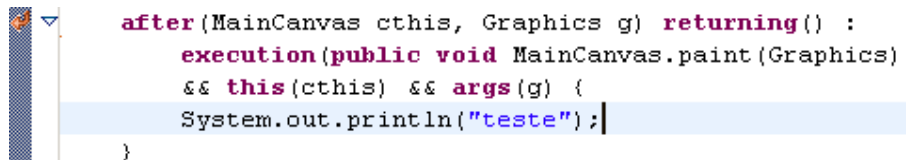


Figura 6.16: código gerado

6.6. Refactoring Change Class Hierarchy

1. Com um arquivo Java™ aberto em um editor, clique no botão *Refactoring Change Class Hierarchy*. A classe principal desse arquivo deve ser subclasse de uma outra classe (Figura 6.17);

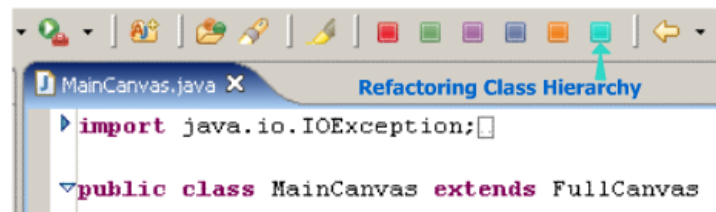


Figura 6.17: arquivo .java aberto

2. Uma caixa de diálogo é aberta, perguntando se você quer gerar o código correspondente ao *refactoring* em um aspecto já existente ou se quer gerar em um novo aspecto (Figura 6.18);

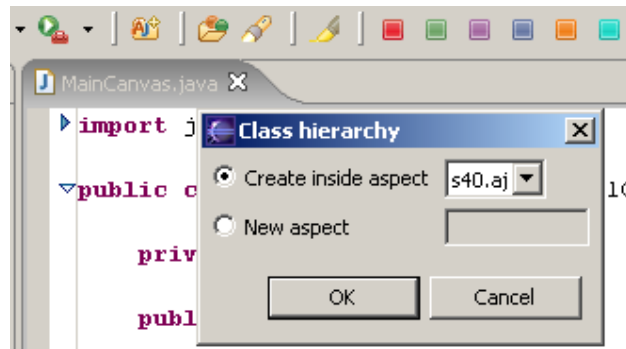


Figura 6.18: seleção de opção

3. A ferramenta executa a verificação das pré-condições no arquivo aberto com o objetivo de garantir que o código gerado vai estar sintaticamente e semanticamente correto. O nome da superclasse é trocado pelo nome da superclasse dessa superclasse (Figura 6.19);

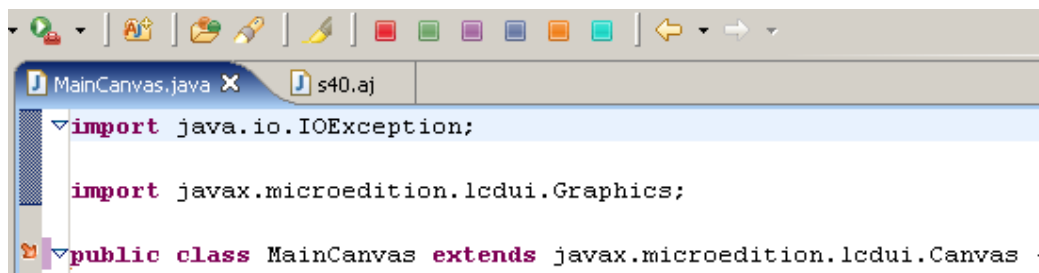


Figura 6.19: arquivo .java alterado

4. O código em *AspectJ* é gerado e escrito em um aspecto já existente ou em um novo aspecto, dentro do mesmo pacote do arquivo .java, conforme a opção escolhida no passo 2 (Figura 6.20).

```
declare parents: MainCanvas extends FullCanvas;
```

Figura 6.20: código gerado

7. Conclusões e Trabalhos Futuros

A Orientação a Aspectos aliada aos conceitos de montagem de Linha de Produtos tem se mostrado uma solução eficiente no processo de *porting* de jogos de um aparelho celular para outro. Entretanto, ainda é possível melhorar esse processo automatizando etapas da aplicação desses conceitos. Por exemplo, através da implementação de ferramentas com operações de *refactorings* que extraem o código da variação para aspectos, baseadas em padrões identificados em estudos de caso da indústria.

Nossa ferramenta automatiza um conjunto desses *refactorings*, que já estão sendo validados na indústria. Não obstante, ela ainda precisa de outros *refactorings* que são necessários para portar os jogos que foram estudados. Além disso, ela pode evoluir para um ambiente que permite a construção de uma Linha de Produtos completa, dentro do próprio *plug-in*, para cada jogo que precisa ser portado pela empresa. Esses são os próximos passos do projeto, que já tem financiamento aprovado por Instituições de Pesquisa para ser concluído.

Dentre todas as contribuições do projeto, a mais importante foi a identificação dos recursos necessários da plataforma Eclipse, do JDT e do AJDT, para implementar as operações de *refactoring*. Esse foi um dos maiores sucessos deste trabalho, pois essas ferramentas possuem pouca ou nenhuma documentação explicando como utilizar esses recursos. O *Visual Studio* da *Microsoft*® seria uma plataforma interessante para o desenvolvimento dessa ferramenta sob esse aspecto, pois ela possui uma documentação completa de todas suas bibliotecas de funcionalidades. Em contrapartida, por ser uma ferramenta paga e por não ter *plug-ins* tão avançados como o AJDT, apoiado pela *IBM*®, para a utilização de *AspectJ*, ela foi descartada. Infelizmente, a ausência de documentação ainda tem um peso muito grande na adoção de tais *softwares* livres, pois exige uma dedicação muito maior do desenvolvedor, além de ser desestimulante em diversas etapas da implementação do projeto. Felizmente, esses problemas foram superados.

No final da implementação de nossa ferramenta, foi lançado o Eclipse 3.1. Atualizamos a mesma de forma que ela seja executável tanto na versão 3.0 como nessa nova versão. Nosso código também está bem comentado e modularizado, facilitando futuras modificações, se necessário, para as próximas versões do Eclipse. É interessante mencionar esse ponto fraco da ferramenta, pois uma simples mudança de versão exigiu alterações específicas do nosso trabalho. Um acontecimento como esse é um presságio para outros eventos semelhantes. Os próximos desenvolvedores desse projeto devem estar cientes desses obstáculos.

Os próximos passos, além da implementação de outros *refactorings* e da adição de novos recursos que permitirão a montagem da Linha de Produtos, são testar a ferramenta em outros cenários à procura de novos *bugs*, inclusive em um jogo por completo, o que não foi feito neste trabalho. Além disso, algumas pré-condições precisam ser re-avaliadas, pois ainda temos *refactorings* muito restritos, o que diminui as possibilidades de aplicação.

Referências Bibliográficas

- [Alves, 2004] ALVES, V. **Identifying Variations in Mobile Devices**. In *Young Researchers Workshop at the GPCE'04*, Vancouver, Canada, October 2004.
- [Alves et al., 2004] ALVES, V.; MATOS, P.; BORBA, P. **An Incremental Aspect-Oriented Product Line Method for J2ME Game Development**. In *Workshop on Managing Variability Consistently in Design and Code at OOPSLA'04*, Vancouver, Canada, October 2004.
- [Alves et al., 2005] ALVES, V.; MATOS, P.; COLE, L.; BORBA, P.; RAMALHO, G. **Extracting and Evolving Mobile Games Product Lines**. In *9th International Software Product Line Conference (SPLC'05)*, volume 3714 of *Lecture Notes in Computer Science*, September 2005. Springer.
- [Alves, 2005] ALVES, V. **Identifying Variations in Mobile Devices**. *Journal of Object Technology*, 4(3):47--52, April 2005.
- [Cardim et al., 2005] ALVES, V.; CARDIM, I.; CARMO, V.; SAMPAIO, P.; DAMASCENO, A.; BORBA, P.; RAMALHO, G. **Comparative Analysis of Porting Strategies in J2ME Games**. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Budapest, Hungary, September 2005. IEEE Press.
- [Carmo, 2005] CARMO, V. **Técnicas para Construção de Linha de Produtos de Jogos Móveis**. Trabalho de Graduação, Centro de Informática, UFPE, Recife, Brasil, Agosto 2005.
- [Clayberg & Rubel, 2004] CLAYBERG, E.; RUBEL, D. **Eclipse: Building Commercial-Quality Plug-Ins**. Addison-Wesley (June 21, 2004).
- [Clements, 2002] CLEMENTS, P.; NORTHROP, M. **Software Product Lines : Practices and Patterns**. Addison-Wesley, 2002.
- [Fowler et al., 1999] FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W.; ROBERTS, D. **Refactoring: Improving the Design of Existing Code**. Object Technology Series. Addison Wesley, 2000.
- [Gamma, 1995] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns**. Addison-Wesley (January, 1995).
- [Kiczales, 1997] KICZALES, G.; LAMPING, J.; MENDHEKAR A.; MAEDA, C.; LOPES, V.; LOINGTIER, J.; IRWIN, J. **Aspect-Oriented Programming**. In *European Conference on Object-Oriented Programming, ECOOP'97*, LNCS 1241, pages 220–242, Finland, June 1997.
- [Sampaio et al., 2004] SAMPAIO, P.; DAMASCENO, A.; ALVES, V.; RAMALHO, G.; BORBA, P. **Porting Games in J2ME: Challenges, Case Study, and Guidelines**. In *III Brazilian Workshop on Games and Digital Entertainment*, October, 2004, Curitiba, Brasil, Agosto 2005.

[Tira Wireless, 2004] Tira Wireless. TiraJump, 2004. Disponível em:
<http://www.tirawireless.com/jump/>.

Datas e Assinaturas

Recife, 20 de agosto de 2005.

Alexandre Torres Vasconcelos
(proponente)

Paulo Henrique Monteiro Borba
(Orientador)

This document was created with Win2PDF available at <http://www.daneprairie.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.